Writing an Ice Application with C-Sharp

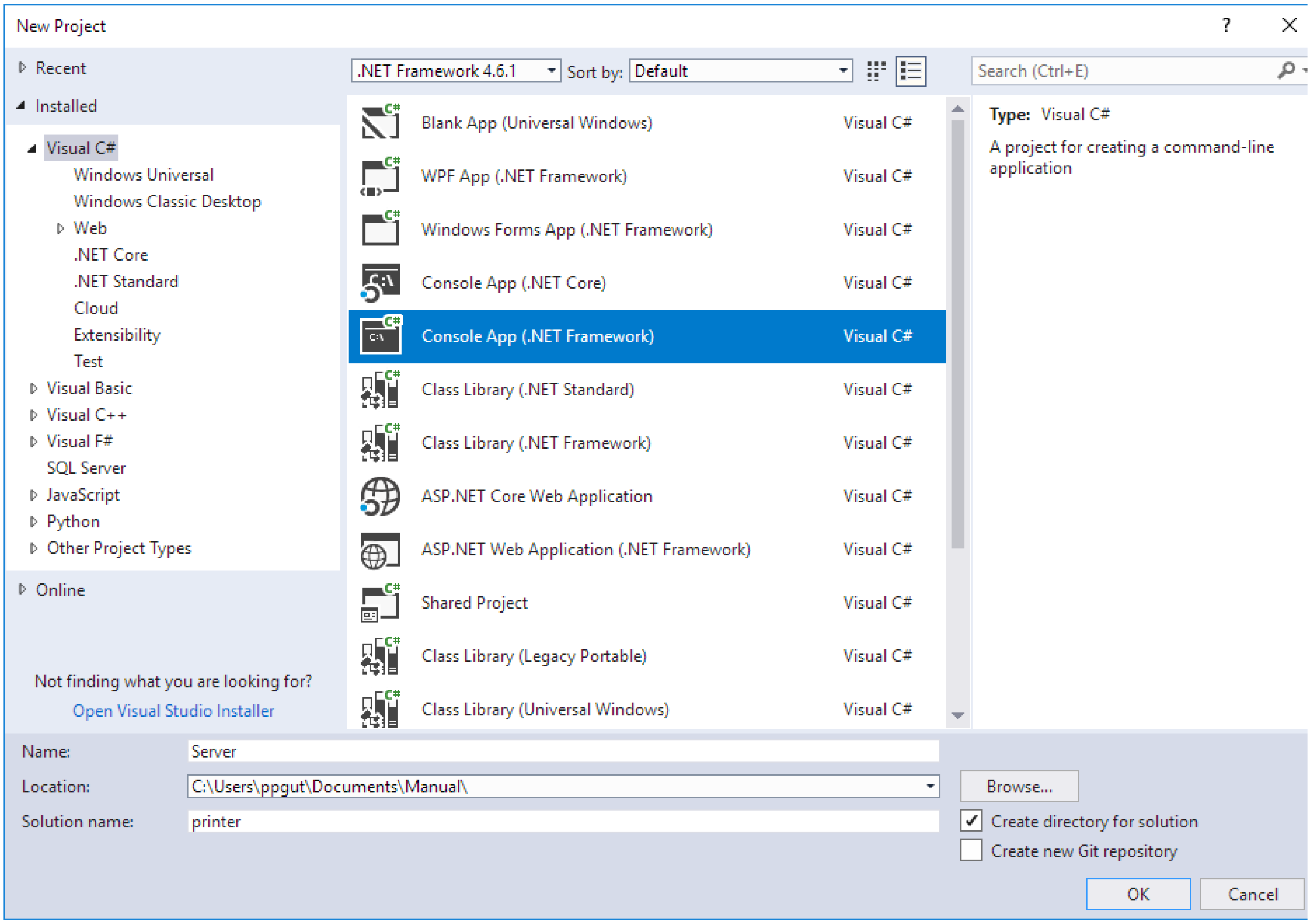
This page shows how to create an Ice application with C#.

On this page:

Create Projects for your Client and Server Applications

We create two projects, one for the Server application and one for the Client application. These are regular Console projects with very little Ice-specific additions.

.NET Framework 4.5 with Visual Studio
Open Visual Studio and create a new Console Application


New Project
?
Recent
Installed
Visual C#
Windows Universal
Windows Classic Desktop
Web
.NET Core
.NET Standard
Cloud
Extensibility
Test
Visual Basic
Visual C++
Visual F#
SQL Server
JavaScript
Python
Other Project Types
Online
Not finding what you are looking for?
Open Visual Studio Installer
Search (Ctrl+E)
Blank App (Universal Windows) Visual C#
WPF App (.NET Framework) Visual C#
Windows Forms App (.NET Framework) Visual C#
Console App (.NET Core) Visual C#
Console App (.NET Framework) Visual C#
Class Library (.NET Standard) Visual C#
Class Library (.NET Framework) Visual C#
ASP.NET Core Web Application Visual C#
ASP.NET Web Application (.NET Framework) Visual C#
Shared Project Visual C#
Class Library (Legacy Portable) Visual C#
Class Library (Universal Windows) Visual C#
Browse...
OK
Cancel

.NET Framework 4.6.1 Sort by: Default

Type: Visual C#

A project for creating a command-line application

Name: Server

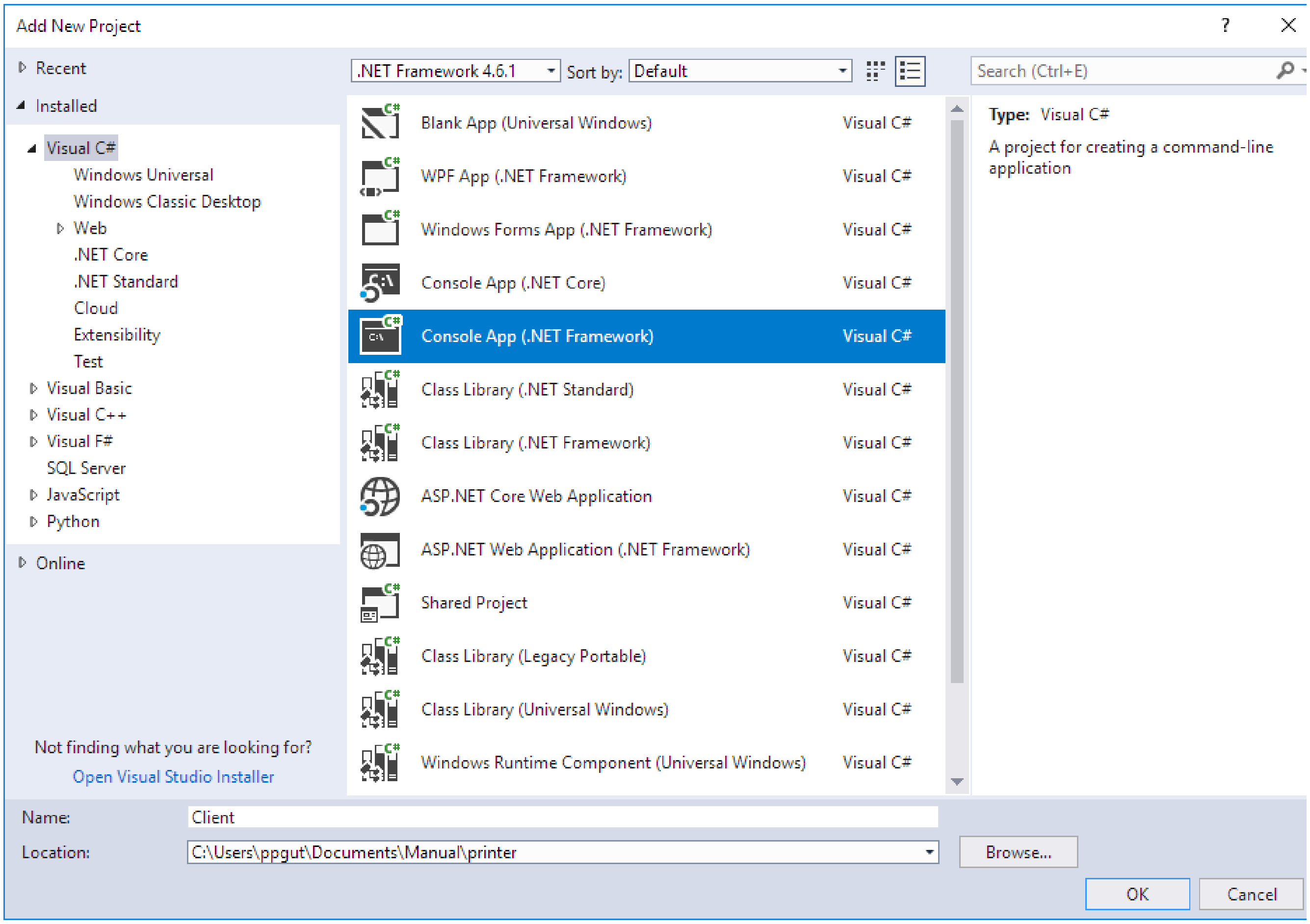
Location: C:\Users\ppgut\Documents\Manual\

Solution name: printer

☑ Create directory for solution

□ Create new Git repository

Create the client project using "File > Add > New Project..."



Add New Project

- Recent
- Installed
  - Visual C#
    - Windows Universal
    - Windows Classic Desktop
    - Web
    - .NET Core
    - .NET Standard
    - Cloud
    - Extensibility
    - Test
  - Visual Basic
  - Visual C++
  - Visual F#
  - SQL Server
  - JavaScript
  - Python
- Online

Not finding what you are looking for?
Open Visual Studio Installer

.NET Framework 4.6.1 Sort by: Default

Blank App (Universal Windows)	Visual C#
WPF App (.NET Framework)	Visual C#
Windows Forms App (.NET Framework)	Visual C#
Console App (.NET Core)	Visual C#
Console App (.NET Framework)	Visual C#
Class Library (.NET Standard)	Visual C#
Class Library (.NET Framework)	Visual C#
ASP.NET Core Web Application	Visual C#
ASP.NET Web Application (.NET Framework)	Visual C#
Shared Project	Visual C#
Class Library (Legacy Portable)	Visual C#
Class Library (Universal Windows)	Visual C#
Windows Runtime Component (Universal Windows)	Visual C#

Search (Ctrl+E)

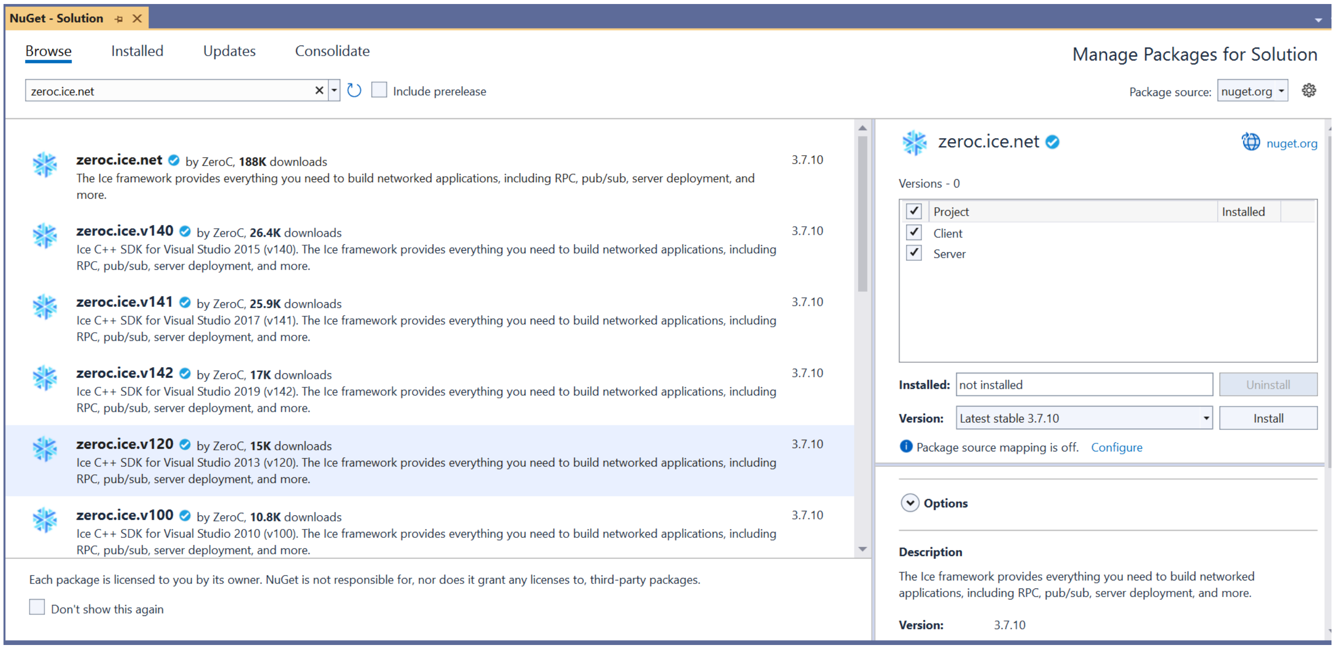
Type: Visual C#

A project for creating a command-line application

Name: Client

Location: C:\Users\ppgut\Documents\Manual\printer

Add the `zeroc.icebuilder.msbuild` and `zeroc.ice.net` NuGet package to the projects with the NuGet Package Manager, found in "Tools > NuGet Package Manager > Manage NuGet Packages for Solution...".



Manage Packages for Solution

zeroc.ice.net ☐ Include prerelease

Package source: nuget.org

zeroc.ice.net by ZeroC, **188K** downloads 3.7.10
The Ice framework provides everything you need to build networked applications, including RPC, pub/sub, server deployment, and more.

zeroc.ice.v140 by ZeroC, **26.4K** downloads 3.7.10
Ice C++ SDK for Visual Studio 2015 (v140). The Ice framework provides everything you need to build networked applications, including RPC, pub/sub, server deployment, and more.

zeroc.ice.v141 by ZeroC, **25.9K** downloads 3.7.10
Ice C++ SDK for Visual Studio 2017 (v141). The Ice framework provides everything you need to build networked applications, including RPC, pub/sub, server deployment, and more.

zeroc.ice.v142 by ZeroC, **17K** downloads 3.7.10
Ice C++ SDK for Visual Studio 2019 (v142). The Ice framework provides everything you need to build networked applications, including RPC, pub/sub, server deployment, and more.

zeroc.ice.v120 by ZeroC, **15K** downloads 3.7.10
Ice C++ SDK for Visual Studio 2013 (v120). The Ice framework provides everything you need to build networked applications, including RPC, pub/sub, server deployment, and more.

zeroc.ice.v100 by ZeroC, **10.8K** downloads 3.7.10
Ice C++ SDK for Visual Studio 2010 (v100). The Ice framework provides everything you need to build networked applications, including RPC, pub/sub, server deployment, and more.

Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses to, third-party packages.

☐ Don't show this again

zeroc.ice.net nuget.org

Versions - 0

☑ Project	Installed
☑ Client	
☑ Server	

Installed: not installed

Version: Latest stable 3.7.10

Package source mapping is off. Configure

Options

Description

The Ice framework provides everything you need to build networked applications, including RPC, pub/sub, server deployment, and more.

Version: 3.7.10

.NET 6.0
Open a new Command Prompt a run the following command to create the server and client projects:

```
dotnet new console -o Server
```

This generates a new .NET Core console application project in the `Server` directory.

Then add references to the `zeroc.icebuilder.msbuild` and `zeroc.ice.net` NuGet packages to this project:

```
dotnet add Server package zeroc.icebuilder.msbuild
dotnet add Server package zeroc.ice.net
```

Finally, repeat these steps for the client project:

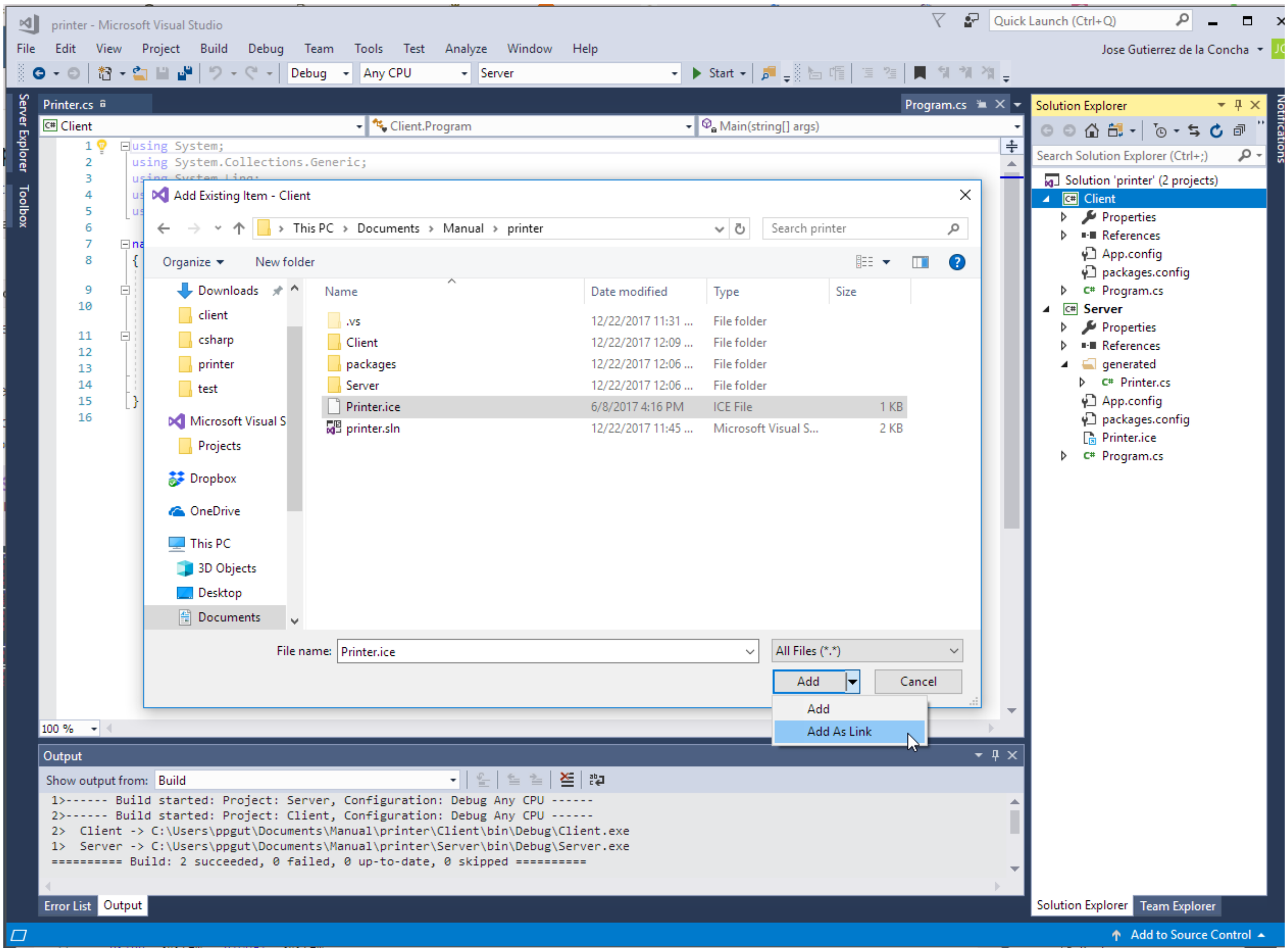
```
dotnet new console -o Client
dotnet add Client package zeroc.icebuilder.msbuild
dotnet add Client package zeroc.ice.net
```

Compile your Slice File

The next step is to add the Slice file (`Printer.ice`) created earlier to each project, and then compile this Slice file.

800px

.NET Framework 4.5 with Visual Studio
Open the "Project > Add Existing Item" dialog and add `Printer.ice` to your Project:



```
using System;
using System.Collections.Generic;
```

Name	Date modified	Type	Size
.vs	12/22/2017 11:31 ...	File folder	
Client	12/22/2017 12:09 ...	File folder	
packages	12/22/2017 12:06 ...	File folder	
Server	12/22/2017 12:06 ...	File folder	
Printer.ice	6/8/2017 4:16 PM	ICE File	1 KB
printer.sln	12/22/2017 11:45 ...	Microsoft Visual S...	2 KB

File name: Printer.ice

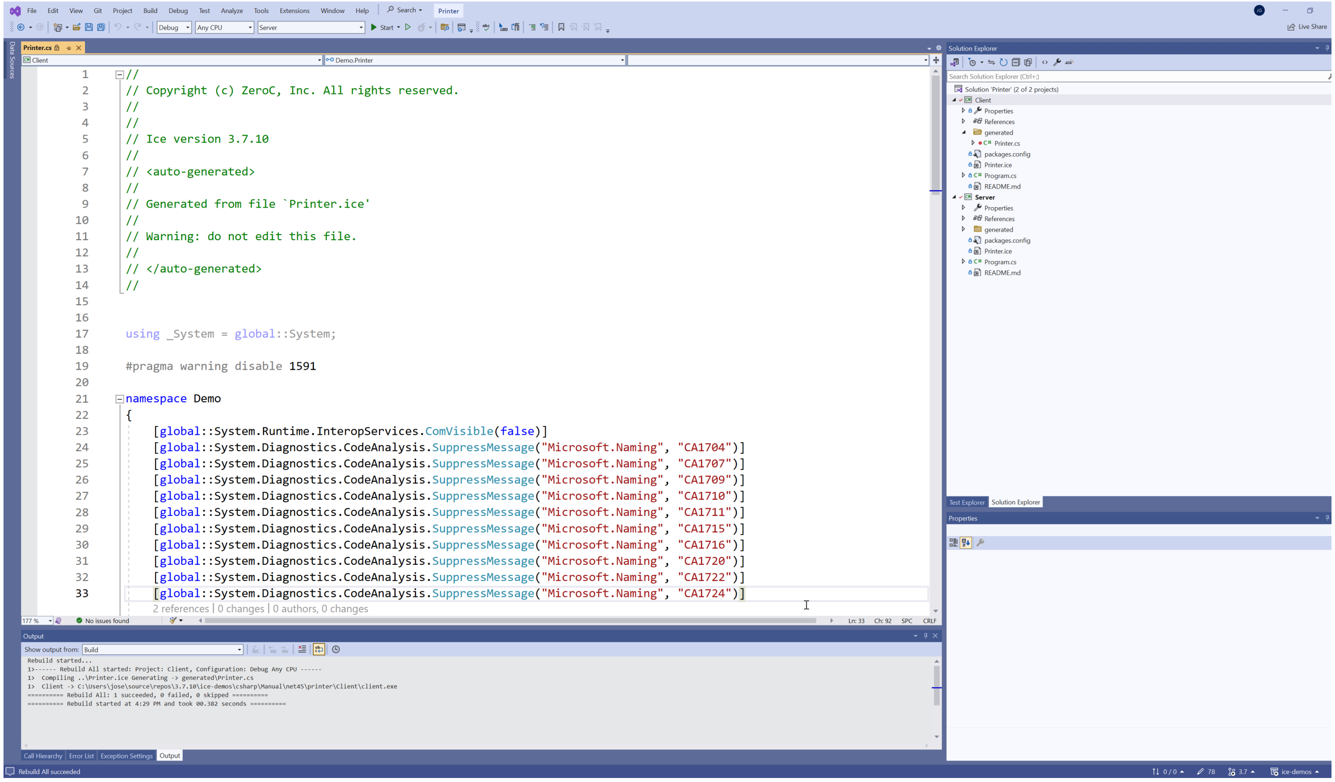
All Files (*.*)

Show output from: Build

```
1>------ Build started: Project: Server, Configuration: Debug Any CPU ------
2>------ Build started: Project: Client, Configuration: Debug Any CPU ------
2>  Client -> C:\Users\ppgut\Documents\Manual\printer\Client\bin\Debug\Client.exe
1>  Server -> C:\Users\ppgut\Documents\Manual\printer\Server\bin\Debug\Server.exe
========== Build: 2 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

If the Ice Builder for Visual Studio is installed, it immediately generates the file `generated\Printer.cs` from `Printer.ice` unless you disabled automatic building in the Ice Builder.

If you have automatic building disabled, select `Build` to build your project. The build generates `generated\Printer.cs` from `Printer.ice` (using Ice Builder) and then compiles both `generated\Printer.cs` and the default no-op `Program.cs`.



```
//
// Copyright (c) ZeroC, Inc. All rights reserved.
//
//
// Ice version 3.7.10
//
// <auto-generated>
//
// Generated from file `Printer.ice'
//
// Warning: do not edit this file.
//
// </auto-generated>
//


using _System = global::System;

#pragma warning disable 1591

namespace Demo
{
    [global::System.Runtime.InteropServices.ComVisible(false)]
    [global::System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Naming", "CA1704")]
    [global::System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Naming", "CA1707")]
    [global::System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Naming", "CA1709")]
    [global::System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Naming", "CA1710")]
    [global::System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Naming", "CA1711")]
    [global::System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Naming", "CA1715")]
    [global::System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Naming", "CA1716")]
    [global::System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Naming", "CA1720")]
    [global::System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Naming", "CA1722")]
    [global::System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Naming", "CA1724")]
    2 references | 0 changes | 0 authors, 0 changes
```

```
Rebuild started...
1>------ Rebuild All started: Project: Client, Configuration: Debug Any CPU ------
1>  Compiling ..\Printer.ice Generating -> generated\Printer.cs
1>  Client -> C:\Users\jose\source\repos\3.7.10\ice-demos\csharp\Manual\net45\printer\Client\client.exe
========== Rebuild All: 1 succeeded, 0 failed, 0 skipped ==========
========== Rebuild started at 4:29 PM and took 00.382 seconds ==========
```

> Ice Builder invokes the Slice to C# compiler (`slice2cs`) to compile Slice files into C# files.

.NET 6.0
Add a Slice item that includes `Printer.ice` to your two projects. The code below shows the client project:

Client.csproj

```
<?xml version="1.0" encoding="utf-8"?>
<Project Sdk="Microsoft.NET.Sdk">
    <PropertyGroup>
        <OutputType>Exe</OutputType>
        <TargetFramework>net6.0</TargetFramework>
    </PropertyGroup>
    <ItemGroup>
        <SliceCompile Include="../Printer.ice" />
        <PackageReference Include="zeroc.ice.net" Version="3.7.8" />
        <PackageReference Include="zeroc.icebuilder.msbuild" Version="5.0.9" />
    </ItemGroup>
</Project>
```

When building the project, the `SliceCompile` task (imported automatically from the `zeroc.icebuilder.msbuild` NuGet package) compiles `Printer.ice` into `generated/Printer.cs` using the Slice to C# compiler, slice2cs.

Use the following command to build the projects:

```
dotnet build Server
dotnet build Client
```

Write and Compile your Server

To implement our `Printer` interface, we must create a servant class. By convention, a servant class uses the name of its interface with an `I`-suffix, so our servant class is called `PrinterI` and we will place it into the default C# source file Program.cs:

C#

```
using System;

namespace Server
{
    public class PrinterI : Demo.PrinterDisp_
    {
        public override void printString(string s, Ice.Current current)
        {
            Console.WriteLine(s);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

The `PrinterI` class inherits from a base class called `PrinterDisp_`, which is generated by the `slice2cs` compiler. The base class is abstract and contains a `printString` method that accepts a string for the printer to print and a parameter of type `Ice.Current`. (For now we will ignore the `Ice.Current` parameter.) Our implementation of the `printString` method simply writes its argument to the terminal.

The remainder of the server code follows in `Program.cs` and is shown in full here:

C#

```
using System;

namespace Server
{
    public class PrinterI : Demo.PrinterDisp_
    {
        public override void printString(string s, Ice.Current current)
        {
            Console.WriteLine(s);
        }
    }

    public class Program
    {
        public static int Main(string[] args)
        {
            try
            {
                using(Ice.Communicator communicator = Ice.Util.initialize(ref args))
                {
                    var adapter =
                        communicator.createObjectAdapterWithEndpoints("SimplePrinterAdapter", "default -h
localhost -p 10000");
                    adapter.add(new PrinterI(), Ice.Util.stringToIdentity("SimplePrinter"));
                    adapter.activate();
                    communicator.waitForShutdown();
                }
            }
            catch(Exception e)
            {
                Console.Error.WriteLine(e);
                return 1;
            }
            return 0;
        }
    }
}
```

The body of `Main` contains a `try` block in which we place all the server code, followed by a `catch` block. The catch block catches all exceptions that may be thrown by the code; the intent is that, if the code encounters an unexpected run-time exception anywhere, the stack is unwound all the way back to `Main`, which prints the exception and then returns failure to the operating system.

The `Ice.Communicator` object implements `IDisposable`, which allows us to use the `using` statement for the initialization of the `Ice.Communicator` object. This ensures the communicator `destroy` method is called when the `using` block goes out of scope. Doing this is essential in order to correctly finalize the Ice run time.

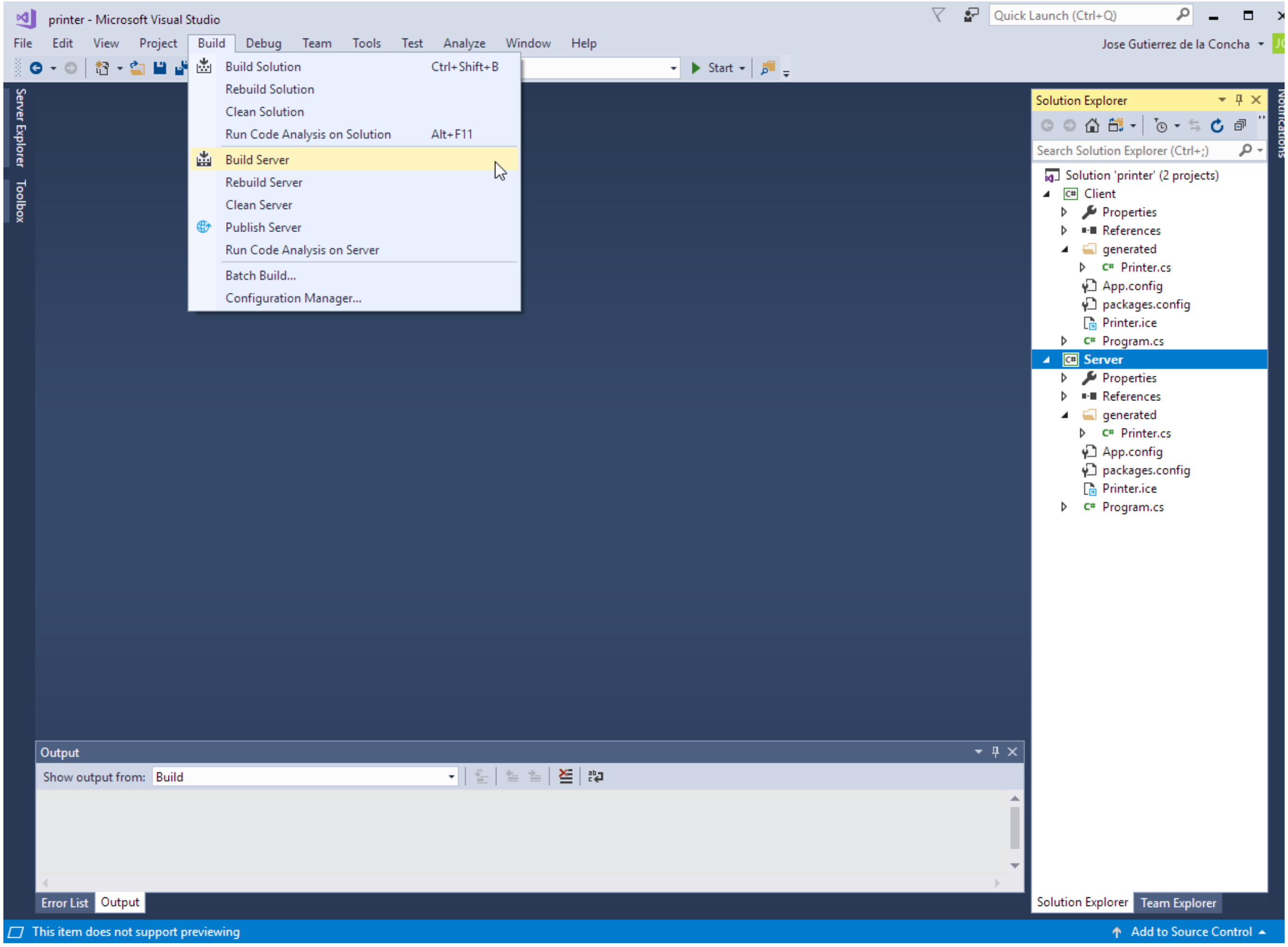
The body of our `try` block contains the actual server code.

The code goes through the following steps:

1. We initialize the Ice run time by calling `Ice.Util.initialize`. (We pass `args` to this call because the server may have command-line arguments that are of interest to the run time; for this example, the server does not require any command-line arguments.) The call to `initialize` returns an `Ice.Communicator` reference, which is the main object in the Ice run time.
2. We create an object adapter by calling `createObjectAdapterWithEndpoints` on the `Communicator` instance. The arguments we pass are `"SimplePrinterAdapter"` (which is the name of the adapter) and `"default -p 10000"`, which instructs the adapter to listen for incoming requests using the default transport protocol (TCP/IP) at port number 10000.
3. At this point, the server-side run time is initialized and we create a servant for our `Printer` interface by instantiating a `PrinterI` object.
4. We inform the object adapter of the presence of a new servant by calling `add` on the adapter; the arguments to `add` are the servant we have just instantiated, plus an identifier. In this case, the string `"SimplePrinter"` is the name of the Ice object. (If we had multiple printers, each would have a different name or, more correctly, a different *object identity*.)
5. Next, we activate the adapter by calling its `activate` method. (The adapter is initially created in a holding state; this is useful if we have many servants that share the same adapter and do not want requests to be processed until after all the servants have been instantiated.)
6. Finally, we call `waitForShutdown`. This call suspends the calling thread until the server implementation terminates, either by making a call to shut down the run time, or in response to a signal. (For now, we will simply interrupt the server on the command line when we no longer need it.)

We can compile the server code as follows:

.NET Framework 4.5 with Visual Studio
Build the server project using "Build > Builder Server"



.NET 6.0
Build the server project using the dotnet `build` command:

```
cd Server
dotnet build
```

Write and Compile your Client

The client code, in `Client/Program.cs`, looks very similar to the server.

Here it is in full:

C#

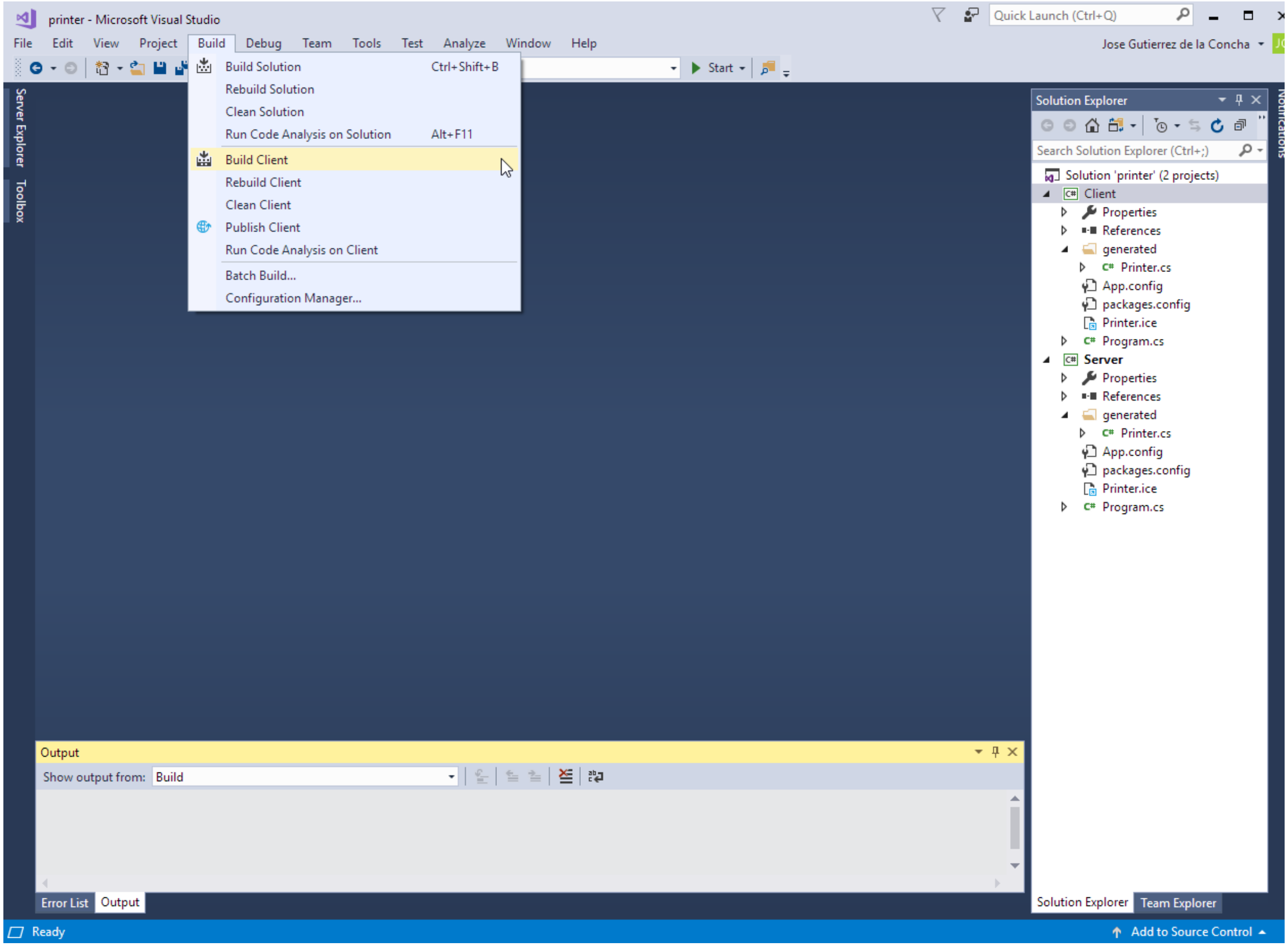
```
using Demo;
using System;

namespace Client
{
    public class Program
    {
        public static int Main(string[] args)
        {
            try
            {
                using(Ice.Communicator communicator = Ice.Util.initialize(ref args))
                {
                    var obj = communicator.stringToProxy("SimplePrinter:default -h localhost -p 10000");
                    var printer = PrinterPrxHelper.checkedCast(obj);
                    if(printer == null)
                    {
                        throw new ApplicationException("Invalid proxy");
                    }

                    printer.printString("Hello World!");
                }
            }
            catch(Exception e)
            {
                Console.Error.WriteLine(e);
                return 1;
            }
            return 0;
        }
    }
}
```

Note that the overall code layout is the same as for the server: we use the same `try` and `catch` blocks to deal with errors. The code in the `try` block does the following:

1. As for the server, we initialize the Ice run time by calling `Ice.Util.initialize` within the `using` statement
2. The next step is to obtain a proxy for the remote printer. We create a proxy by calling `stringToProxy` on the communicator, with the string `"SimplePrinter:default -p 10000"`. Note that the string contains the object identity and the port number that were used by the server. (Obviously, hard-coding object identities and port numbers into our applications is a bad idea, but it will do for now; we will see more architecturally sound ways of doing this when we discuss IceGrid.
3. The proxy returned by `stringToProxy` is of type `Ice.ObjectPrx`, which is at the root of the inheritance tree for interfaces and classes. But to actually talk to our printer, we need a proxy for a `Printer` interface, not an `Object` interface. To do this, we need to do a down-cast by calling `PrinterPrxHelper.checkedCast`. A checked cast sends a message to the server, effectively asking "is this a proxy for a `Printer` interface?" If so, the call returns a proxy of type `Demo::Printer`; otherwise, if the proxy denotes an interface of some other type, the call returns null.
4. We test that the down-cast succeeded and, if not, throw an error message that terminates the client.
5. We now have a live proxy in our address space and can call the `printString` method, passing it the time-honored `"Hello World!"` string. The server prints that string on its terminal.

The client's project is just like the server's project shown earlier.

.NET Framework 4.5 with Visual Studio
Build the client project using "Build > Builder Client"



.NET 6.0

Build the client project using dotnet `build` command:

```
cd Client
dotnet build
```

Run your Client and Server

To run client and server, we first start the server in a separate window:

.NET Framework 4.5

```
server
```

.NET 6.0

```
cd Server
dotnet run
```

At this point, we won't see anything because the server simply waits for a client to connect to it. We run the client in a different window:

.NET Framework 4.5

```
client
```

.NET 6.0

```
cd Client
dotnet run
```

The client runs and exits without producing any output; however, in the server window, we see the `"Hello World!"` that is produced by the printer. To get rid of the server, we just interrupt it on the command line for now.

If anything goes wrong, the client will print an error message. For example, if we run the client without having first started the server, we get something like the following:

```
Ice.ConnectionRefusedException
    error = 0
```

See Also

- Client-Side Slice-to-C-Sharp Mapping
- Server-Side Slice-to-C-Sharp Mapping
- The Current Object
- IceGrid