

# Contents

## Deployment documentation

### Overview

- [First look at deployment](#)

- [Deploy to a folder, IIS, or Azure](#)

### Quickstarts

- [Deploy to Azure App Service](#)

- [Deploy to App Service for Linux](#)

- [Deploy to a web site](#)

- [Deploy to a folder](#)

### Tutorials

#### .NET

- [Deploy a .NET Windows application using ClickOnce](#)

- [Deploy a .NET Framework desktop app using ClickOnce](#)

- [Create an installer package \(Windows desktop\)](#)

- [Package a desktop app for Microsoft Store >>](#)

- [Build ClickOnce applications from the Command Line](#)

#### ASP.NET

- [Publish an ASP.NET Core app to Azure >>](#)

- [Import publish settings and deploy to IIS](#)

- [Import publish settings and deploy to Azure](#)

- [Continuous deployment of ASP.NET Core to Azure with Git >>](#)

#### C++

- [Create an installer package](#)

- [Package a desktop app for Microsoft Store >>](#)

- [Deploy a C++/CLR app using ClickOnce >>](#)

#### UWP

- [Package a UWP app by using Visual Studio >>](#)

#### Node.js

- [Publish to Linux App Service >>](#)

[Python](#)

[Publish to Azure App Service >>](#)

## [How-to guides](#)

### [ClickOnce security and deployment](#)

[Overview of ClickOnce security and deployment](#)

[Choose a ClickOnce deployment strategy](#)

[ClickOnce cache overview](#)

[ClickOnce and application settings](#)

[ClickOnce deployment on Windows Vista](#)

### [Localization](#)

[Localize ClickOnce applications](#)

[Publish a Project that has a specific locale](#)

### [Security](#)

[Secure ClickOnce applications](#)

[ClickOnce and Authenticode](#)

[Trusted application deployment overview](#)

[Code access security for ClickOnce applications](#)

[Enable ClickOnce security settings](#)

[Set a security zone for a ClickOnce application](#)

[Set custom permissions for a ClickOnce application](#)

[Add a trusted publisher to a client computer](#)

[Re-sign application and deployment manifests](#)

[Configure the ClickOnce trust prompt behavior](#)

[Sign setup files with SignTool.exe \(ClickOnce\)](#)

### [Publish](#)

[Publish ClickOnce applications](#)

[Publish a ClickOnce application using the Publish Wizard](#)

[Create ClickOnce applications for others to deploy](#)

[Deploy apps for test and production servers without resigning](#)

[Access local and remote data in ClickOnce applications](#)

[Deploy COM components with ClickOnce](#)

[Build ClickOnce applications from the command line](#)

Specify where Visual Studio copies the files

Specify the location where end users will install from

Specify the ClickOnce offline or online install mode

Set the ClickOnce publish version

Automatically increment the ClickOnce publish version

Specify files

- Specify which files are published by ClickOnce

- Include a data file in a ClickOnce application

Install prerequisites with a ClickOnce application

Include prerequisites with a ClickOnce application

Manage updates for a ClickOnce application

Change the publish language for a ClickOnce application

Specify a Start Menu name for a ClickOnce application

Specify Technical Support

- Specify a link for Technical Support

- Specify a support URL for individual prerequisites

Publish Page

- Specify a publish page for a ClickOnce application

- Customize the default web page for a ClickOnce application

Enable AutoStart for CD Installations

Create file associations For a ClickOnce application

Retrieve query string information in an online ClickOnce application

Disable URL activation

- Disable URL activation of ClickOnce applications by using the Designer

- Disable URL activation of ClickOnce applications

Deploy apps that can run on multiple versions of the .NET Framework

Publish a WPF application with visual styles enabled

Download Assemblies on Demand

- Walkthrough: Download assemblies on demand using the Designer

- Walkthrough: Download assemblies on demand

Walkthrough: Download satellite assemblies on demand using the Designer

Manual Deployment

Walkthrough: Manually deploy a ClickOnce application

Walkthrough: Manually deploy an app that does not require re-signing

Walkthrough: Download satellite assemblies on demand

Walkthrough: Create a custom installer

## Update Strategy

Choose a ClickOnce update strategy

How ClickOnce performs application updates

Check for application updates programmatically

Specify an alternate location for deployment updates

ClickOnce deployment samples and walkthroughs

## Troubleshooting

Troubleshoot ClickOnce deployments

Set a custom log file location for ClickOnce deployment errors

Specify verbose log files for ClickOnce deployments

Server and client configuration issues in ClickOnce deployments

Security, versioning, and manifest issues in ClickOnce deployments

Troubleshoot specific errors in ClickOnce deployments

Debug ClickOnce applications that use System.Deployment.Application

## Application deployment prerequisites

Overview of application deployment prerequisites

Deploy prerequisites for 64-bit applications

Create bootstrapper packages

Create a product manifest

Create a package manifest

Create a localized bootstrapper package

Walkthrough: Create a custom bootstrapper with a privacy prompt

## Product and Package Schema Reference

Overview of Product and Package Schema Reference

<Product> Element (Bootstrapper)

<Package> Element (Bootstrapper)

<RelatedProducts> Element (Bootstrapper)

<InstallChecks> Element (Bootstrapper)

- [<Commands> Element \(Bootstrapper\)](#)
- [<PackageFiles> Element \(Bootstrapper\)](#)
- [<Strings> Element \(Bootstrapper\)](#)
- [<Schedules> Element \(Bootstrapper\)](#)

## Reference

### ClickOnce Reference

- [Overview of ClickOnce Reference](#)

#### ClickOnce application Manifest

- [Overview of ClickOnce application Manifest](#)
- [<assembly> Element \(ClickOnce application\)](#)
- [<assemblyIdentity> Element \(ClickOnce application\)](#)
- [<trustInfo> Element \(ClickOnce application\)](#)
- [<entryPoint> Element \(ClickOnce application\)](#)
- [<dependency> Element \(ClickOnce application\)](#)
- [<file> Element \(ClickOnce application\)](#)
- [<fileAssociation> Element \(ClickOnce application\)](#)

#### ClickOnce Deployment Manifest

- [Overview of ClickOnce Deployment Manifest](#)
- [<assembly> Element \(ClickOnce Deployment\)](#)
- [<assemblyIdentity> Element \(ClickOnce Deployment\)](#)
- [<description> Element \(ClickOnce Deployment\)](#)
- [<deployment> Element \(ClickOnce Deployment\)](#)
- [<compatibleFrameworks> Element \(ClickOnce Deployment\)](#)
- [<dependency> Element \(ClickOnce Deployment\)](#)
- [<publisherIdentity> Element \(ClickOnce Deployment\)](#)
- [<Signature> Element \(ClickOnce Deployment\)](#)
- [<customErrorReporting> Element \(ClickOnce Deployment\)](#)

#### ClickOnce Unmanaged API Reference

### Visual Studio Installer Projects Reference

- [Visual Studio Installer Projects Extension and .NET Core 3.1](#)

# First look at deployment in Visual Studio

3/5/2021 • 5 minutes to read • [Edit Online](#)

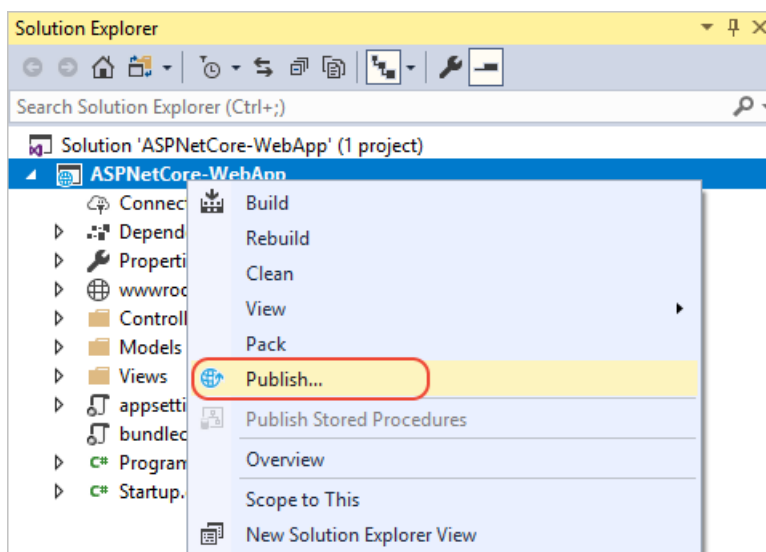
By deploying an application, service, or component, you distribute it for installation on other computers, devices, or servers, or in the cloud. You choose the appropriate method in Visual Studio for the type of deployment that you need. (Many app types support other deployment tools, such as command-line deployment or NuGet, that aren't described here.)

See the quickstarts and tutorials for step-by-step deployment instructions. For an overview of deployment options, see [What publishing options are right for me?](#).

## Deploy to a local folder

Deployment to a local folder is typically used for testing or to begin a staged deployment in which another tool is used for final deployment.

- **ASP.NET, ASP.NET Core, Node.js, Python, and .NET Core:** Use the **Publish** tool to deploy to a local folder. The exact options available depend on your app type. In Solution Explorer, right-click your project and select **Publish**. (If you haven't previously configured any publishing profiles, you must then select **Create new profile**.) Next, select **Folder**. For more information, see [Deploy to a local folder](#).



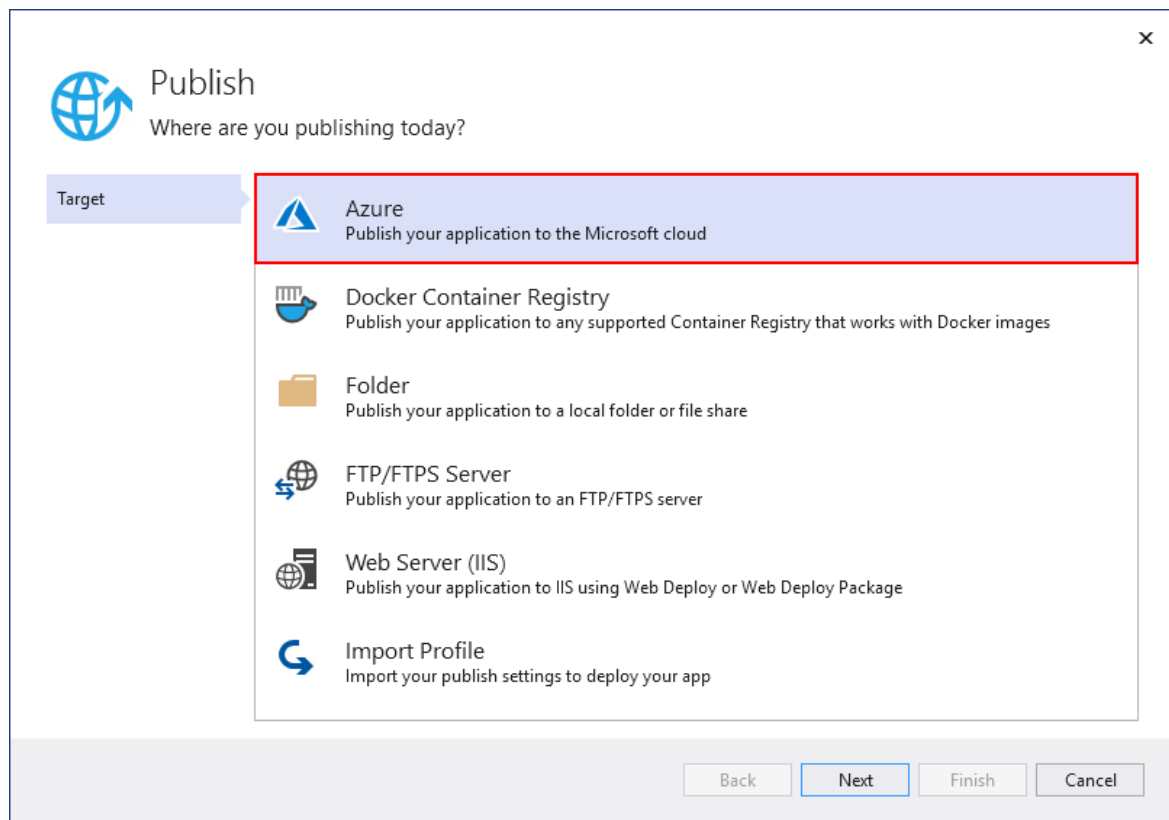
- **Windows desktop:** You can publish a Windows desktop application to a folder by using ClickOnce deployment. Users can then install the application with a single click. For more information, see the following articles:
  - [Deploy a .NET Framework Windows desktop app using ClickOnce](#).
  - [Deploy a .NET Windows desktop app using ClickOnce](#).
  - [Deploy a C++/CLR app using ClickOnce](#) or, for C/C++, see [Deploy a native app using a Setup project](#).

## Publish to Azure

- **ASP.NET, ASP.NET Core, Python, and Node.js:** Publish to Azure App Service or Azure App Service on Linux (using containers) by using one of the following methods:
  - For continuous (or automated) deployment of apps, use Azure DevOps with [Azure Pipelines](#).
  - For one-time (or manual) deployment of apps, use the **Publish** tool in Visual Studio.

For deployment that provides more customized configuration of the server, you can also use the **Publish** tool to deploy apps to an Azure virtual machine.

To use the **Publish** tool, right-click the project in Solution Explorer and select **Publish**. (If you've previously configured any publishing profiles, you must then select **Create new profile**.) In the **Publish** dialog box, select either **App Service** or **Azure Virtual Machines**, and then follow the configuration steps.



Starting in Visual Studio 2017 version 15.7, you can deploy ASP.NET Core apps to App Service on Linux.

For Python apps, also see [Python - Publishing to Azure App Service](#).

For a quick introduction, see [Publish to Azure](#) and [Publish to Linux](#). Also, see [Publish an ASP.NET Core app to Azure](#). For deployment using Git, see [Continuous deployment of ASP.NET Core to Azure with Git](#).

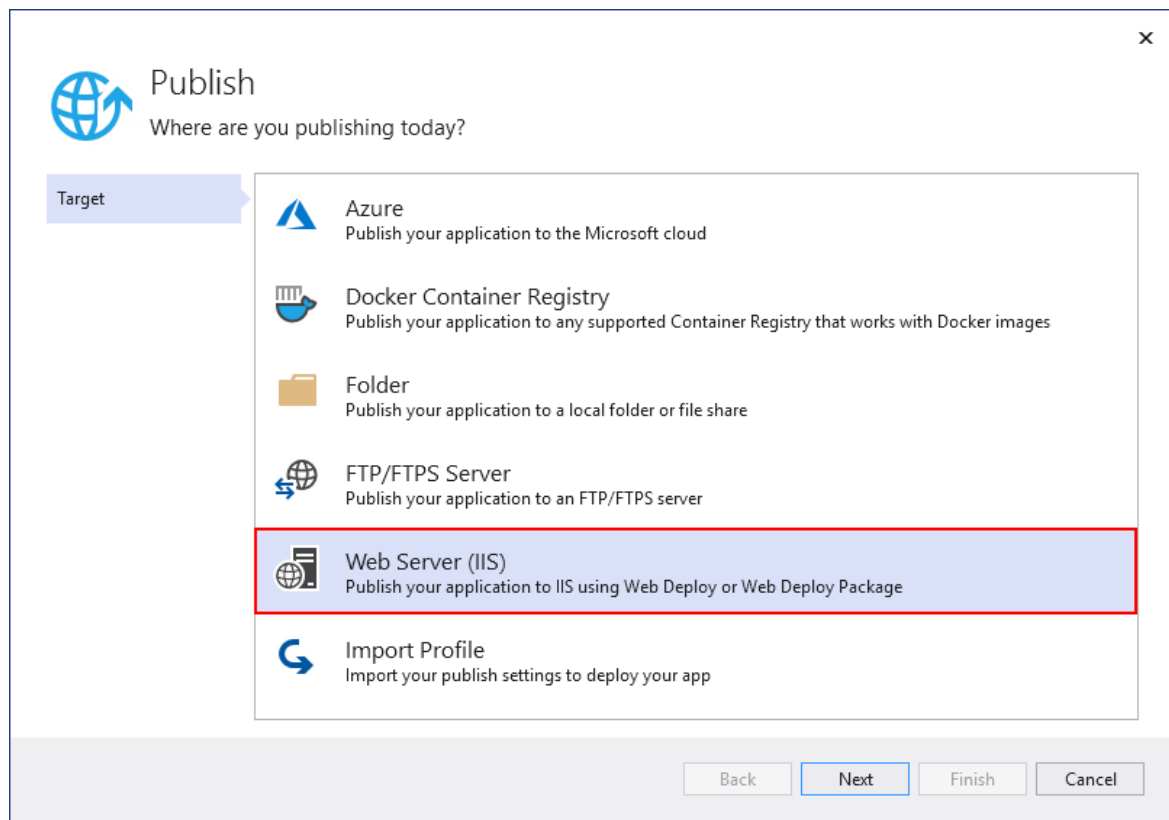
#### NOTE

If you don't already have an Azure account, you can [sign up here](#).

## Publish to the web or deploy to a network share

- **ASP.NET, ASP.NET Core, Node.js, and Python:** You can use the **Publish** tool to deploy to a website by using FTP or Web Deploy. For more information, see [Deploy to a website](#).

In Solution Explorer, right-click the project and select **Publish**. (If you've previously configured any publishing profiles, you must then select **Create new profile**.) In the **Publish** tool, select the option you want and follow the configuration steps.



For information on importing a publish profile in Visual Studio, see [Import publish settings and deploy to IIS](#).

You can also deploy ASP.NET applications and services in a number of other ways. For more information, see [Deploying ASP.NET web applications and services](#).

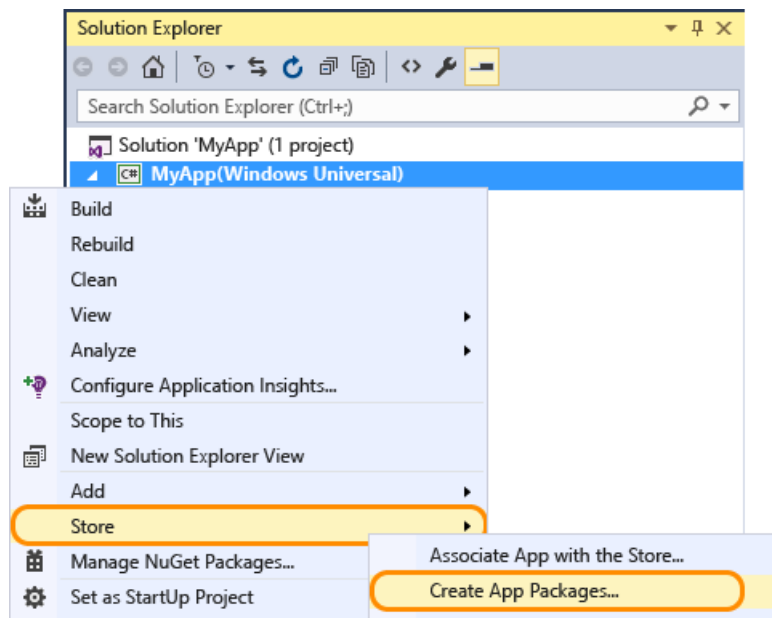
- **Windows desktop:** You can publish a Windows desktop application to a web server or a network file share by using ClickOnce deployment. Users can then install the application with a single click. For more information, see the following articles:
  - [Deploy a .NET Framework Windows desktop app using ClickOnce](#)
  - [Deploy a .NET Windows desktop app using ClickOnce](#)
  - [Deploy a C++/CLR app using ClickOnce](#)

## Publish to Microsoft Store

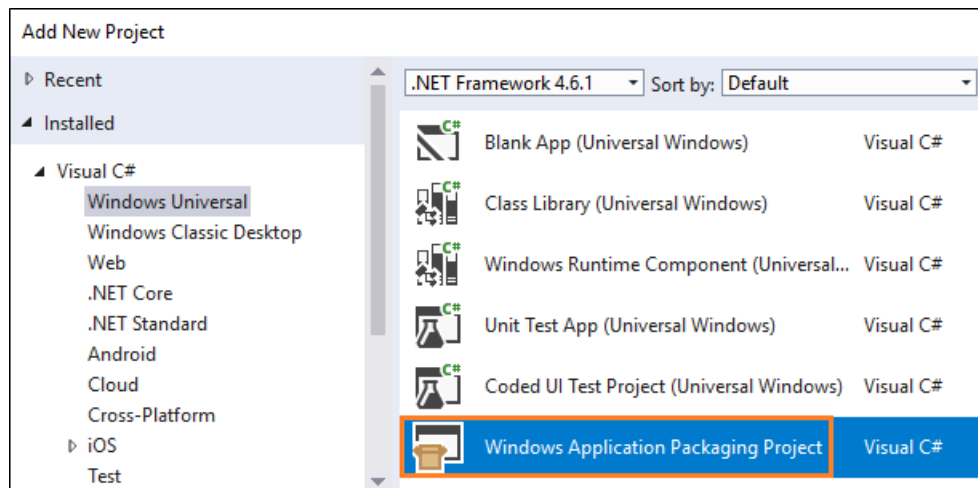
From Visual Studio, you can create app packages for deployment to Microsoft Store.

- **UWP:** You can package your app and deploy it by using menu items. For more information, see [Package a UWP app by using Visual Studio](#).





- **Windows desktop:** You can deploy to Microsoft Store by using the Desktop Bridge starting in Visual Studio 2017 version 15.4. To do this, start by creating a Windows Application Packaging Project. For more information, see [Package a desktop app for Microsoft Store \(Desktop Bridge\)](#).



## Deploy to a device (UWP)

If you're deploying a UWP app for testing on a device, see [Run UWP apps on a remote machine in Visual Studio](#).

## Create an installer package (Windows desktop)

If you require a more complex installation of a desktop application than ClickOnce can provide, you can create a Windows Installer package (MSI or EXE installation file) or a custom bootstrapper.

- An MSI-based installer package can be created by using the [WiX Toolset Visual Studio 2017 Extension](#). This is a command-line toolset.
- An MSI or EXE installer package can be created by using [InstallShield](#) from Flexera Software. InstallShield may be used with Visual Studio 2017 and later versions. Community Edition isn't supported.

### NOTE

InstallShield Limited Edition is no longer included with Visual Studio and isn't supported in Visual Studio 2017 and later versions. Check with [Flexera Software](#) about future availability.

- An MSI or EXE installer package can be created by using a Setup project (vdproj). To use this option, install the [Visual Studio Installer Projects extension](#).
- You can also install prerequisite components for desktop applications by configuring a generic installer, which is known as a bootstrapper. For more information, see [Application deployment prerequisites](#).

## Deploy to a test lab

You can enable more sophisticated development and testing by deploying your applications into virtual environments. For more information, see [Test on a lab environment](#).

## Continuous deployment

You can use Azure Pipelines to enable continuous deployment of your app. For more information, see [Azure Pipelines](#) and [Deploy to Azure](#).

## Deploy a SQL database

- [Change target platform and publish a database project \(SQL Server Data Tools \(SSDT\)\)](#)
- [Deploy an Analysis Services Project \(SSAS\)](#)
- [Deploy Integration Services \(SSIS\) projects and packages](#)
- [Build and deploy to a local database](#)

## Deployment for other app types

| APP TYPE             | DEPLOYMENT SCENARIO   | LINK   |
|----------------------|---|--|
| Office app           | You can publish an add-in for Office from Visual Studio.  | <a href="#">Deploy and publish your Office add-in</a>      |
| WCF or OData service | Other applications can use WCF RIA services that you deploy to a web server.  | <a href="#">Developing and deploying WCF Data Services</a> |
| LightSwitch          | LightSwitch is no longer supported starting in Visual Studio 2017, but can still be deployed from Visual Studio 2015 and earlier. | <a href="#">Deploying LightSwitch applications</a>         |

## Next steps

In this tutorial, you took a quick look at deployment options for different applications.

[What publishing options are right for me?](#)

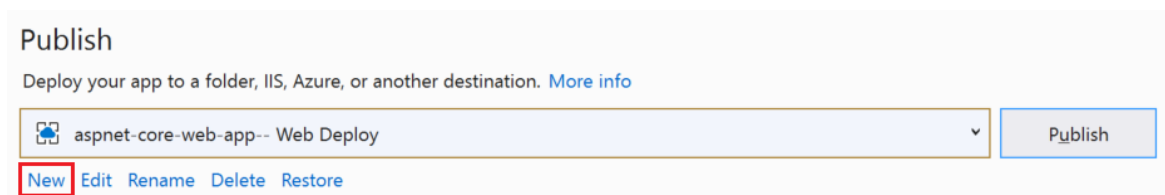
# Deploy your app to a folder, IIS, Azure, or another destination

3/5/2021 • 9 minutes to read • [Edit Online](#)

By deploying an application, service, or component, you distribute it for installation on other computers, devices, servers, or in the cloud. You choose the appropriate method in Visual Studio for the type of deployment that you need.

Get help for your deployment task:

- Not sure what deployment option to choose? See [What publishing options are right for me?](#)
- For help with deployment issues for Azure App Service or IIS, see [Troubleshoot ASP.NET Core on Azure App Service and IIS](#).
- For help with configuring .NET deployment settings, see [Configure .NET deployment settings](#).
- To deploy to a new target, if you have previously created a publish profile, select **New** from the **Publish** window for a configured profile.



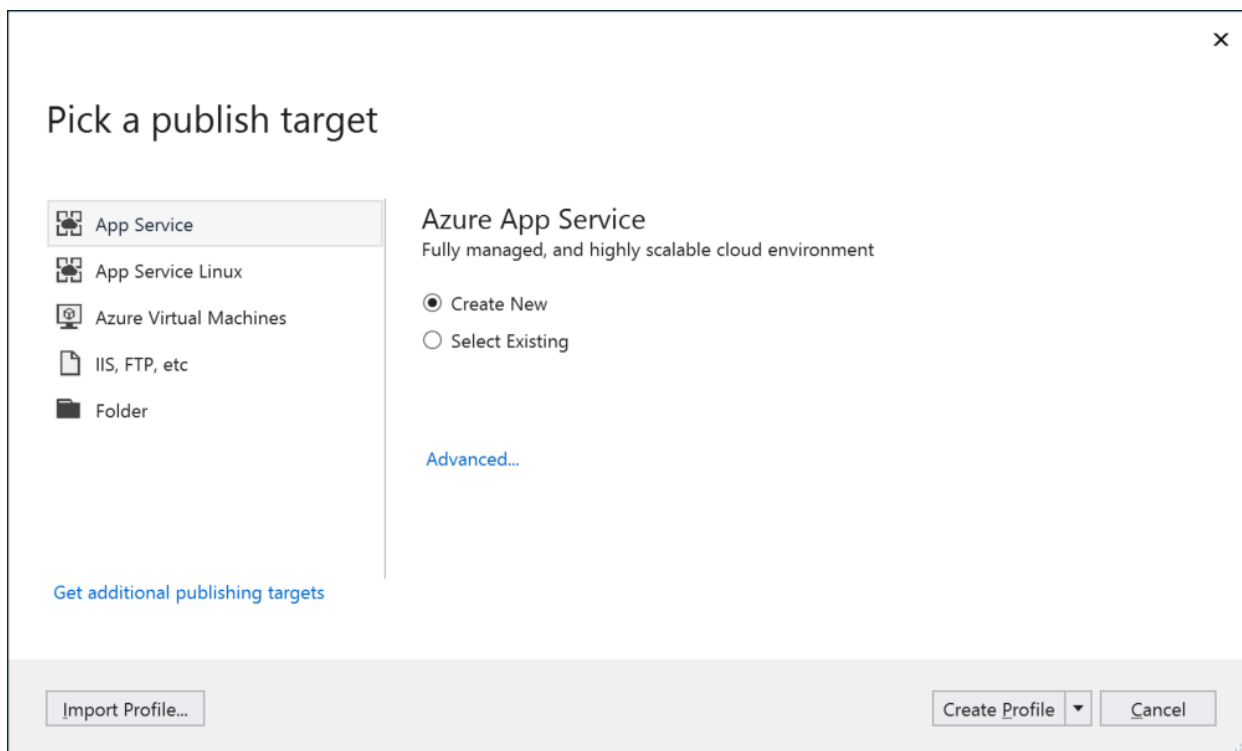
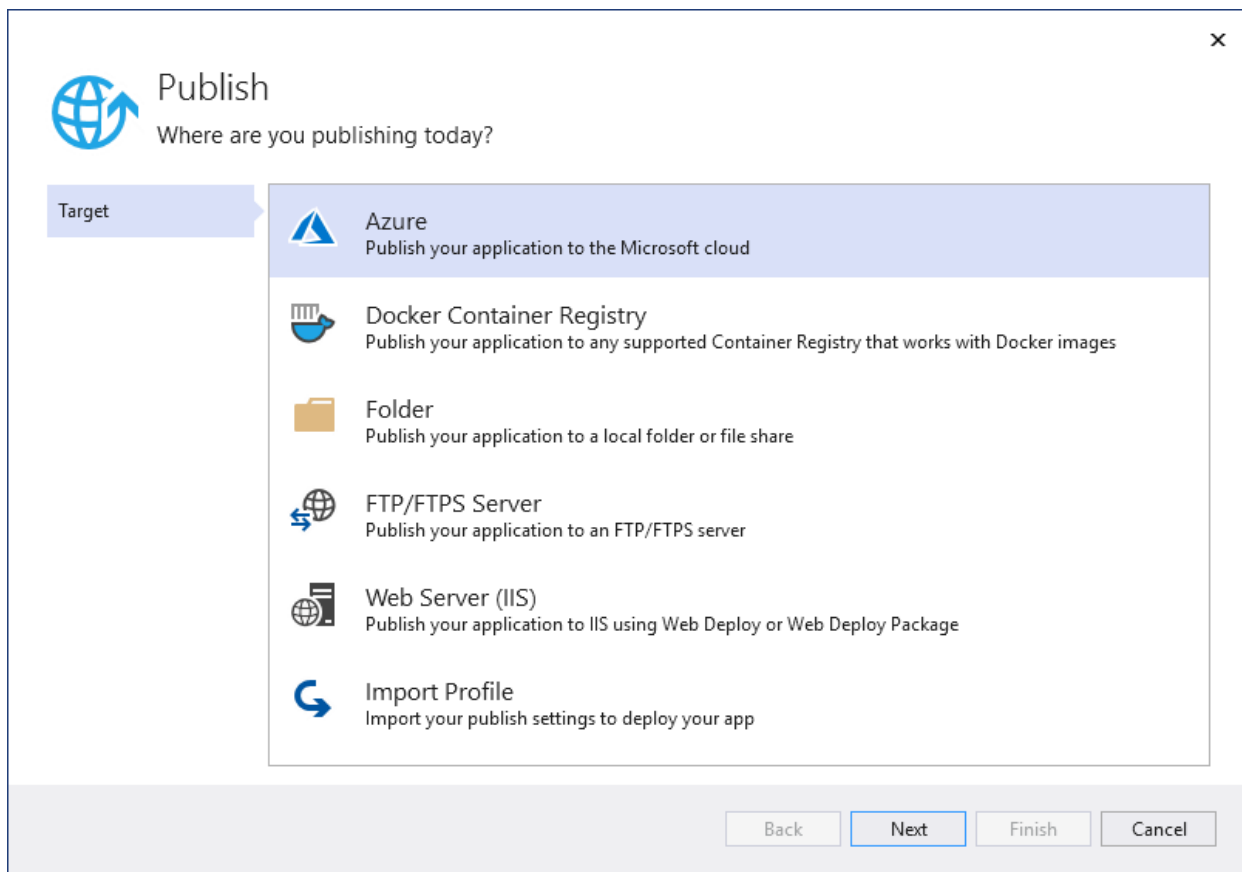
Then, choose a deployment option in the Publish window. For information on your publishing options, see the following sections.

## What publishing options are right for me?

From within Visual Studio, applications can be published directly to the following targets:

- [Azure](#)
- [Docker Container Registry](#)
- [Folder](#)
- [FTP/FTPS server](#)
- [Web server\(IIS\)](#)
- [Import profile](#)
- [App Service](#)
- [App Service Linux](#)
- [IIS \(choose IIS, FTP, etc.\)](#)
- [FTP/FTPS \(choose IIS, FTP, etc.\)](#)
- [Folder](#)
- [Import profile](#)

The preceding options appear as shown in the following illustration when you create a new publish profile.

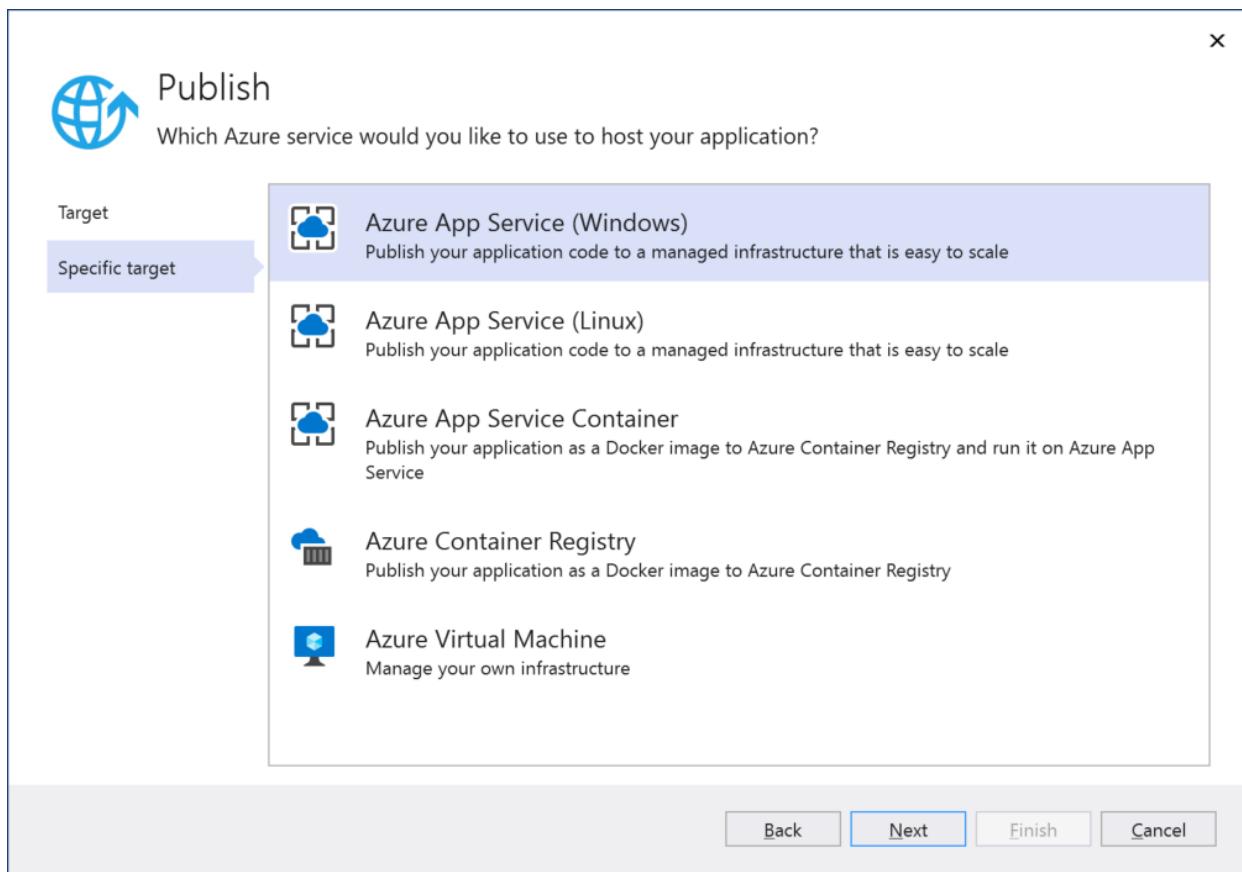


For a quick tour of more general application deployment options, see [First look at deployment](#).

## Azure

When you choose Azure, you can choose between:

- [Azure App Service](#) running on Windows, Linux, or as a Docker image
- A Docker image deployed to [Azure Container Registry](#)
- An [Azure Virtual Machine](#)



## Azure App Service

[Azure App Service](#) helps developers quickly create scalable web applications and services without maintaining infrastructure. An App Service runs on cloud-hosted virtual machines in Azure, but those virtual machines are managed for you. Each app in an App Service will be assigned a unique \*.azurewebsites.net URL; all pricing tiers other than Free allow assigning custom domain names to the site.

You determine how much computing power an App Service has by choosing a [pricing tier or plan](#) for the containing App Service. You can have multiple Web apps (and other app types) share the same App Service without changing the pricing tier. For example, you can host development, staging, and production Web apps together on the same App Service.

### When to choose Azure App Service

- You want to deploy a web application that's accessible through the Internet.
- You want to automatically scale your web application according to demand without needing to redeploy.
- You don't want to maintain server infrastructure (including software updates).
- You don't need any machine-level customizations on the servers that host your web application.

If you want to use Azure App Service in your own datacenter or other on-premises computers, you can do so using the [Azure Stack](#).

For more information on publishing to App Service, see:

- [Quickstart - Publish to Azure App Service](#)
- [Quickstart - Publish ASP.NET Core to Linux](#).
- [Publish an ASP.NET Core app to Azure App Service](#)
- [Troubleshoot ASP.NET Core on Azure App Service and IIS](#).

## Azure Container Registry

[Azure Container Registry](#) allows you to build, store, and manage Docker container images and artifacts in a private registry for all types of container deployments.

#### When to choose Azure Container Registry

- When you have an existing Docker container development and deployment pipeline.
- When you want to build Docker container images in Azure.

For more information:

- [Deploy an ASP.NET container to a container registry](#)

#### Azure Virtual Machine

[Azure Virtual Machines \(VMs\)](#) lets you create and manage any number of computing resources in the cloud. By assuming responsibility for all software and updates on the VMs, you can customize them as much as desired as required by your application. You can access the virtual machines directly through Remote Desktop, and each one will maintain its assigned IP address as long as desired.

Scaling an application that's hosted on virtual machines involves spinning up additional VMs according to demand and then deploying the necessary software. This additional level of control lets you scale differently in different global regions. For example, if your application is serving employees in a variety of regional offices, you can scale your VMs according to the number of employees in those regions, potentially reducing costs.

For additional information, see the [detailed comparison](#) between Azure App Service, Azure Virtual Machines, and other Azure services that you can use as a deployment target using the Custom option in Visual Studio.

#### When to choose Azure Virtual Machines

- You want to deploy a web application that's accessible through the Internet, with full control over the lifetime of assigned IP addresses.
- You need machine-level customizations on your servers, which include additional software such as a specialized database system, specific networking configurations, disk partitions, and so forth.
- You want a fine level of control over scaling of your web application.
- You need direct access to the servers hosting your application for any other reason.

If you want to use Azure Virtual Machines in your own datacenter or other on-premises computers, you can do so using the [Azure Stack](#).

## Docker container registry

If your application is using Docker, you can publish your containerized application to a Docker container registry.

#### When to choose Docker Container Registry

- You want to deploy a containerized application

For more information, see the following:

- [Deploy an ASP.NET container to a container registry](#)
- [Deploy to Docker Hub](#)

## Folder

Deploying to the file system means to copy your application's files to a specific folder on your own computer. Deploying to a folder is most often used for testing purposes, or to deploy the application for use by a limited number of people if the computer is also running a server. If the target folder is shared on a network, then deploying to the file system can make the web application files available to others who might then deploy it to specific servers.

Starting with Visual Studio 2019 16.8, the folder target includes the ability to publish a .Net Windows application using ClickOnce.

If you wish to publish a .NET Core 3.1, or newer, Windows application with ClickOnce, see [Deploy a .NET Windows application using ClickOnce](#).

Any local machines that are running a server can make your application available through the Internet or an Intranet depending on how it's configured and the networks to which it's connected. (If you do connect a computer directly to the Internet, be especially careful to protect it from external security threats.) Because you manage these machines, you're in complete control of the software and hardware configurations.

If for any reason (such as machine access) you are not able to use cloud services like Azure App Service or Azure Virtual Machines, you can use the [Azure Stack](#) in your own datacenter. The Azure Stack allows you to manage and use computing resources through Azure App Service and Azure Virtual Machines while yet keeping everything on-premises.

### When to choose file system deployment

- You need only deploy the application to a file share from which others will deploy it to different servers.
- You want to deploy a .NET Windows Application using ClickOnce
- You need only a local test deployment.
- You want to examine and potentially modify the application files independently before sending them onto another deployment target.

For more information, see [Quickstart - Deploy to a local folder](#).

For more information on deploying a .NET Windows Application using ClickOnce, see [Deploy a .NET Windows application using ClickOnce](#).

For additional help to choose your settings, see the following:

- [Framework-dependent vs. self-contained deployment](#)
- [Target runtime identifiers \(portable RID, et al\)](#)
- [Debug and release configurations](#)

## FTP/FTPS server

An FTP/FTPS server lets you deploy your application to a server other than Azure. It can deploy to a file system or any other server (Internet or Intranet) to which you have access, including those on other cloud services. It can work with web deploy (files or .ZIP) and FTP.

When choosing a FTP/FTPS server, Visual Studio prompts you for a profile name, and then collects additional **Connection** information including the target server or location, a site name, and credentials. You can control the following behaviors on the **Settings** tab:

- The configuration you want to deploy.
- Whether to remove existing files from the destination.
- Whether to precompile during publishing.
- Whether to exclude files in the App\_Data folder from deployment.

You can create any number of FTP/FTPS deployment profiles in Visual Studio, making it possible to manage profiles with different settings.

### When to choose FTP/FTPS server deployment

- You're using cloud services on a provider other than Azure that can be accessed through URLs.
- You want to deploy using credentials other than the ones that you use within Visual Studio, or those tied directly to your Azure accounts.
- You want to delete files from the target each time you deploy.

# Web Server (IIS)

An IIS web server lets you deploy your application to a web server other than Azure. It can deploy to an IIS server (Internet or Intranet) to which you have access, including those on other cloud services. It can work with Web Deploy or a Web Deploy package.

When choosing an IIS web server, Visual Studio prompts you for a profile name, and then collects additional **Connection** information including the target server or location, a site name, and credentials. You can control the following behaviors on the **Settings** tab:

- The configuration you want to deploy.
- Whether to remove existing files from the destination.
- Whether to precompile during publishing.
- Whether to exclude files in the App\_Data folder from deployment.

You can create any number of IIS web server deployment profiles in Visual Studio, making it possible to manage profiles with different settings.

## When to choose web server (IIS) deployment

- You're using IIS to publish a site or service that can be accessed through URLs.
- You want to deploy using credentials other than the ones that you use within Visual Studio, or those tied directly to your Azure accounts.
- You want to delete files from the target each time you deploy.

For more information, see [Quickstart - Deploy to a web site](#).

For help with troubleshooting ASP.NET Core on IIS, see [Troubleshoot ASP.NET Core on Azure App Service and IIS](#).

# Import Profile

You can import a profile when publishing to IIS or Azure App Service. You can configure deployment using a *publish settings file* (*\*.publishsettings*). A publish settings file is created by IIS or Azure App Service, or it can be manually created, and then it can be imported into Visual Studio.

Use of a publish settings file can simplify deployment configuration and works better in a team environment versus manually configuring each deployment profile.

## When to choose import profile

- You're publishing to IIS and want to simplify deployment configuration.
- You're publishing to IIS or Azure App Service and want to speed up deployment configuration for reuse or for team members publishing to the same service.

For more information, see the following:

- [Import publish settings and deploy to IIS](#)
- [Import publish settings and deploy to Azure](#)

# Configure .NET deployment settings

For additional help to choose your settings, see the following:

- [Framework-dependent vs. self-contained deployment](#)
- [Target runtime identifiers \(portable RID, et al\)](#)
- [Debug and release configurations](#)



# Next steps

Tutorials:

- [Deploy a .NET Core application with the publish tool](#)
- [Publish an ASP.NET core app to Azure](#)
- [Deployment in Visual C++](#)
- [Deploy UWP apps](#)
- [Publish a Node.js app to Azure using Web Deploy](#)
- [Publish a Python app to Azure App Service](#)

# Publish a Web app to Azure App Service using Visual Studio

3/5/2021 • 3 minutes to read • [Edit Online](#)

For ASP.NET, ASP.NET Core, Node.js, and .NET Core apps, publish to Azure App Service or Azure App Service Linux (using containers) using one of the following methods.

- For continuous (or automated) deployment of apps, use Azure DevOps with [Azure Pipelines](#).
- For one-time (or manual) deployment of apps, use the **Publish** tool in Visual Studio to deploy ASP.NET, ASP.NET Core, Node.js, and .NET Core apps to Azure App Service or [App Service for Linux](#) (using containers). For Python apps, follow the steps on [Python - Publish to Azure App Service](#).

This article describes how to use the **Publish** tool for one-time deployment.

## Prerequisites

- [Visual Studio 2019](#) installed with the appropriate workloads for your language of choice:
  - ASP.NET: **ASP.NET and web development**
  - Node.js: **Node.js development**
- [Visual Studio 2017](#) installed with the appropriate workloads for your language of choice:
  - ASP.NET: **ASP.NET and web development**
  - Node.js: **Node.js development**
- An Azure subscription. If you do not already have subscription, [sign up for free](#), which includes \$200 in credit for 30 days and 12 months of popular free services.
- An ASP.NET, ASP.NET Core, .NET Core, or Node.js project. If you don't already have a project, select an option below:
  - ASP.NET Core: Follow [Quickstart: Use Visual Studio to create your first ASP.NET Core web app](#), or use the following steps:

In Visual Studio 2019, choose **Create a new project** in the start window. If the start window is not open, choose **File > Start Window**. Type **web app** in the search box, choose **C#** as the language, then choose **ASP.NET Core Web Application (Model-View-Controller)**, and then choose **Next**. On the next screen, name the project **MyASPApp**, and then choose **Next**.

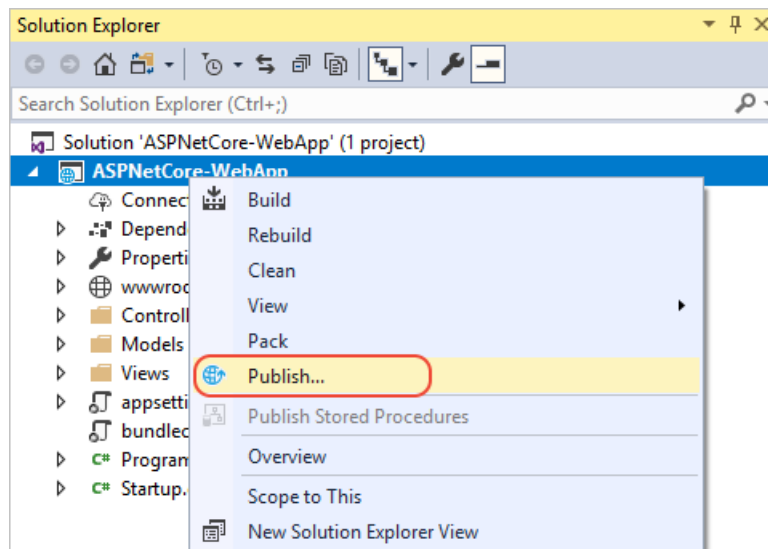
Choose either the recommended target framework (.NET Core 3.1) or .NET 5, and then choose **Create**.

In Visual Studio 2017, choose **File > New Project**, select **Visual C# > .NET Core**, then select **ASP.NET Core Web Application**. When prompted, select the **Web Application (Model-View-Controller)** template, make sure that **No Authentication** is selected, and then select **OK**.

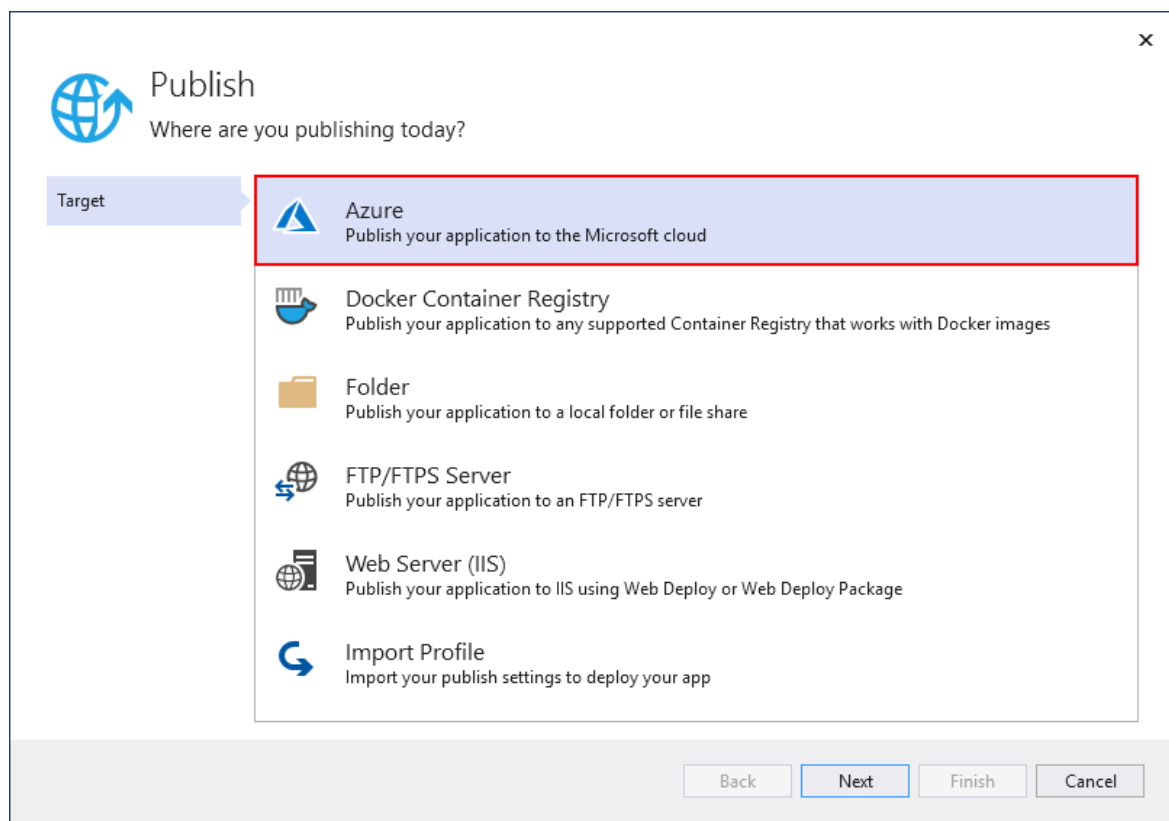
- Node.js: Follow [Quickstart: Use Visual Studio to create your first Node.js app](#), or use **File > New Project**, select **JavaScript**, then select **Blank Node.js Web Application**.
- Make sure you build the project using the **Build > Build Solution** menu command before following the deployment steps.

# Publish to Azure App Service on Windows

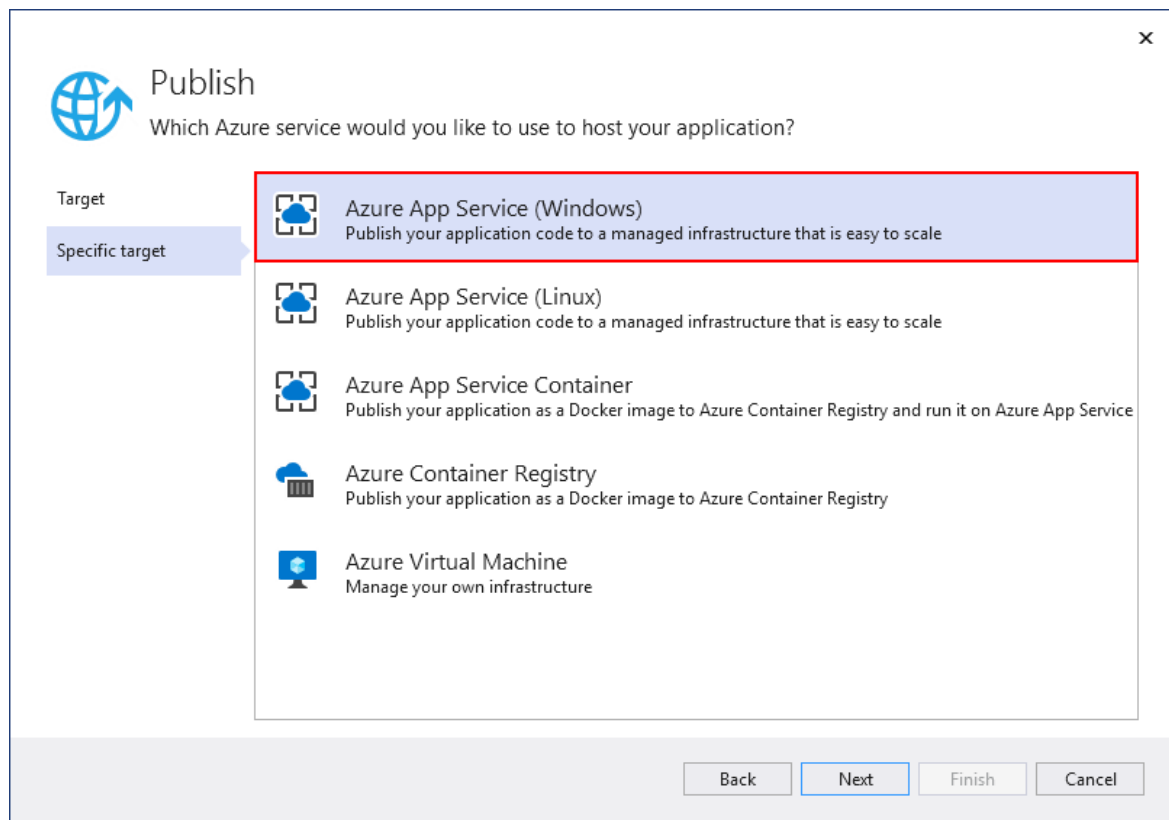
1. In Solution Explorer, right-click the project node and choose **Publish** (or use the **Build** > **Publish** menu item).



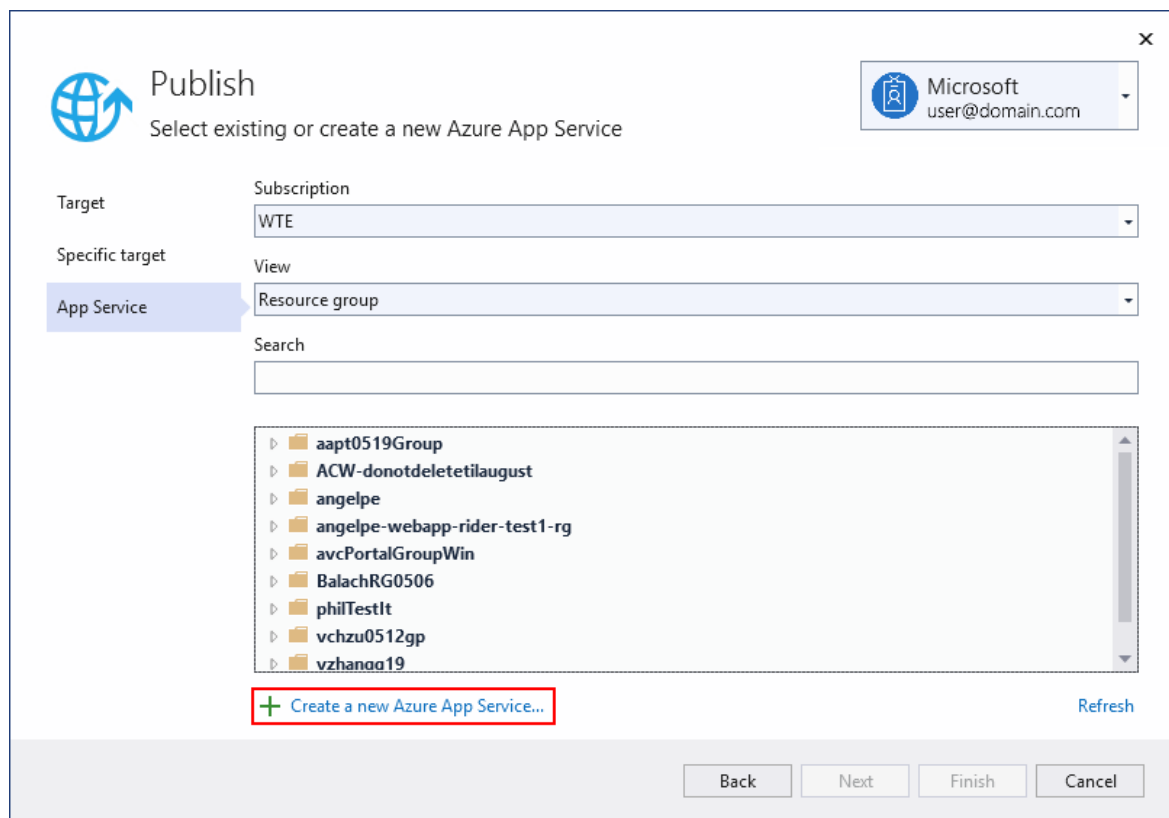
2. If you have previously configured any publishing profiles, the **Publish** window appears. Select **New**.
3. In the **Publish** window, select **Azure**.



4. Select **Azure App Service (Windows)** and **Next**.



5. Sign in with your Azure account, if necessary. Select **Create a new Azure App Service...**



6. In the **Create Azure App Service (Windows)** dialog, the **App Name**, **Resource Group**, and **App Service Plan** entry fields are populated. You can keep these names or change them. When ready, select **Create**.

**App Service (Windows)**  
Create new

Microsoft  
user@domain.com

Name  
MyAspCoreWebAppOnAzure2

Subscription  
WTE

Resource group  
angelpe (West Central US) [New...](#)

Hosting Plan  
angelpe-hp (Central US, S1) [New...](#)

Export... Create Cancel

7. In the **Publish** dialog, the newly created instance has been automatically selected. When ready, select **Finish**.

**Publish**  
Select existing or create a new Azure App Service

Microsoft  
user@domain.com

Target  
Subscription  
WTE

Specific target  
View  
Resource group

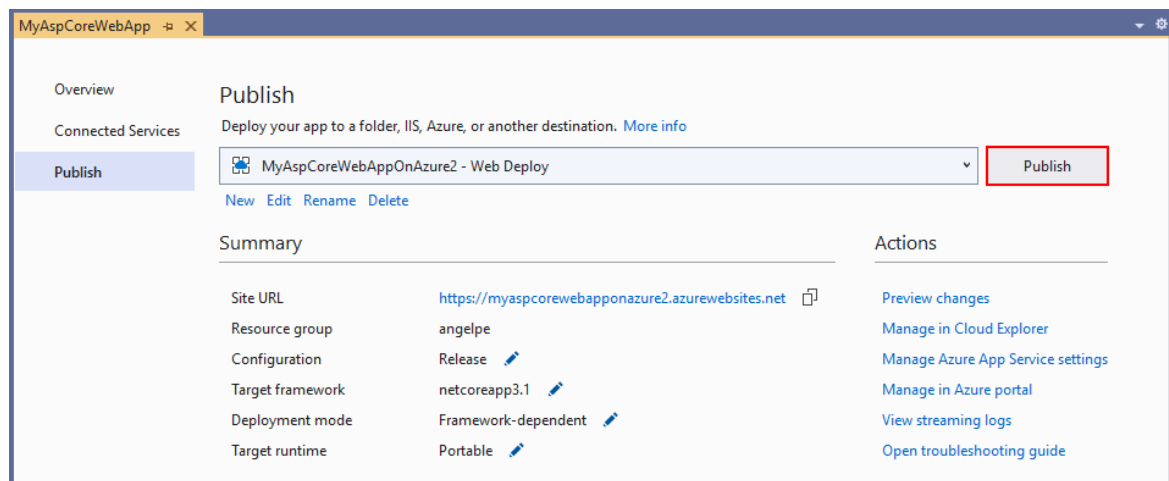
App Service  
Search  
MyAspCoreWebAppOnAzure2

angelpe  
MyAspCoreWebAppOnAzure2  
Deployment Slots

+ Create a new Azure App Service... Refresh

Back Next Finish Cancel

8. Select **Publish**. Visual Studio deploys the app to your Azure App Service, and the web app loads in your browser. The project properties **Publish** pane shows the site URL and other details.



## Clean up resources

In the preceding steps, you created Azure resources in a resource group. If you don't expect to need these resources in the future, you can delete them by deleting the resource group. From the left menu in the Azure portal, select **Resource groups** and then select **myResourceGroup**. On the resource group page, make sure that the listed resources are the ones you want to delete. Select **Delete**, type **myResourceGroup** in the text box, and then select **Delete**.

## Next steps

In this quickstart, you learned how to use Visual Studio to create a publishing profile for deployment to Azure. You can also configure a publishing profile by importing publish settings from Azure App Service.

[Import publish settings and deploy to Azure](#)

# Publish an ASP.NET Core app to App Service on Linux using Visual Studio

3/5/2021 • 2 minutes to read • [Edit Online](#)

Starting in Visual Studio 2017 version 15.7, you can publish ASP.NET Core apps to Azure App Service Linux (using containers) using one of the following methods.

- For continuous (or automated) deployment of apps, use Azure DevOps with [Azure Pipelines](#).
- For one-time (or manual) deployment of apps, use the **Publish** tool in Visual Studio to publish ASP.NET Core apps to App Service for Linux (using containers).

This article describes how to use the **Publish** tool for one-time deployment.

## Prerequisites

- [Visual Studio 2019](#) installed with the appropriate workloads for your language of choice:
  - ASP.NET: **ASP.NET and web development**
- [Visual Studio 2017](#) installed with the appropriate workloads for your language of choice:
  - ASP.NET: **ASP.NET and web development**
- An Azure subscription. If you do not already have subscription, [sign up for free](#), which includes \$200 in credit for 30 days and 12 months of popular free services.
- ASP.NET Core: Follow [Quickstart: Use Visual Studio to create your first ASP.NET Core web app](#), or use the following steps:

In Visual Studio 2019, choose **Create a new project** in the start window. If the start window is not open, choose **File > Start Window**. Type **web app** in the search box, choose **C#** as the language, then choose **ASP.NET Core Web Application (Model-View-Controller)**, and then choose **Next**. On the next screen, name the project **MyASPApp**, and then choose **Next**.

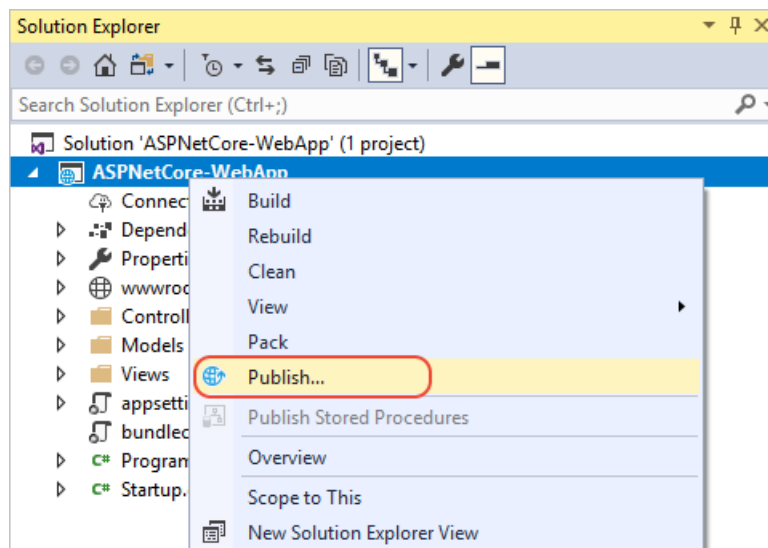
Choose either the recommended target framework (.NET Core 3.1) or .NET 5, and then choose **Create**.

In Visual Studio 2017, choose **File > New Project**, select **Visual C# > .NET Core**, then select **ASP.NET Core Web Application**. When prompted, select the **Web Application (Model-View-Controller)** template, make sure that **No Authentication** is selected, and then select **OK**.

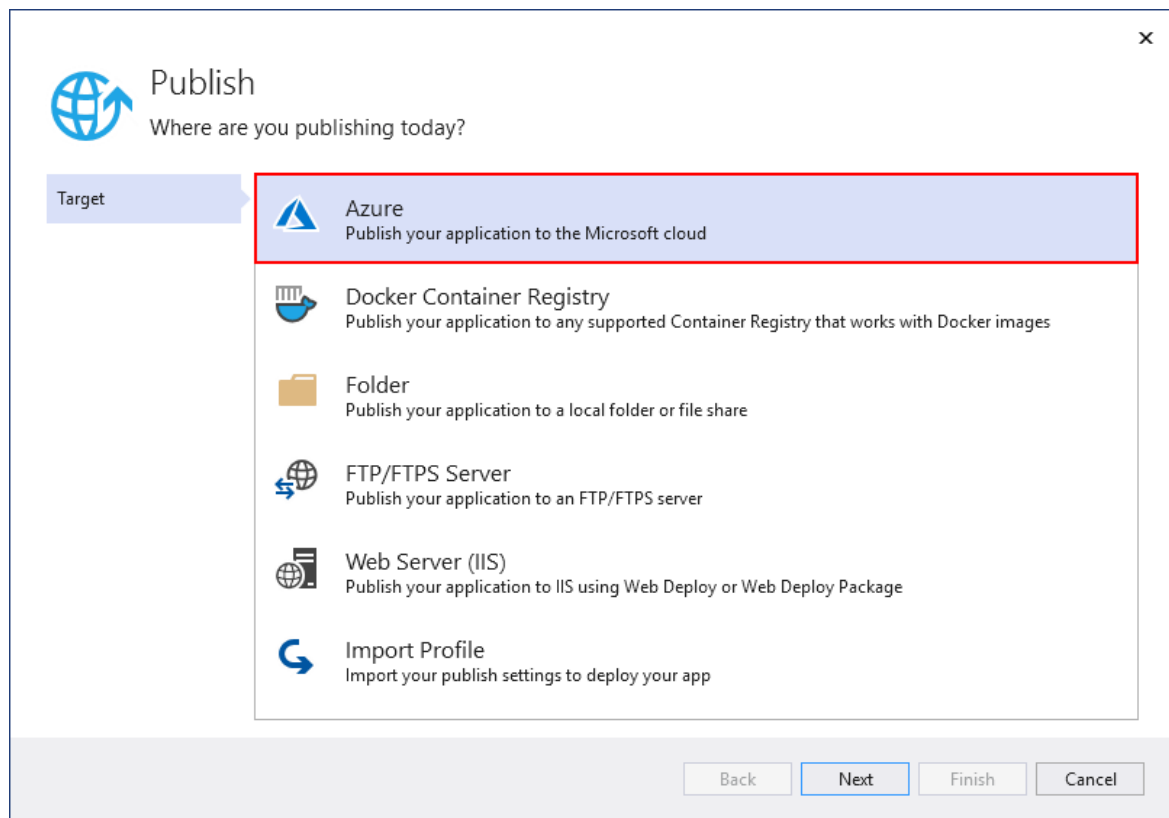
- Make sure you build the project using the **Build > Build Solution** menu command before following the deployment steps.

## Publish to Azure App Service on Linux

1. In Solution Explorer, right-click the project and choose **Publish** (or use the **Build > Publish** menu item).

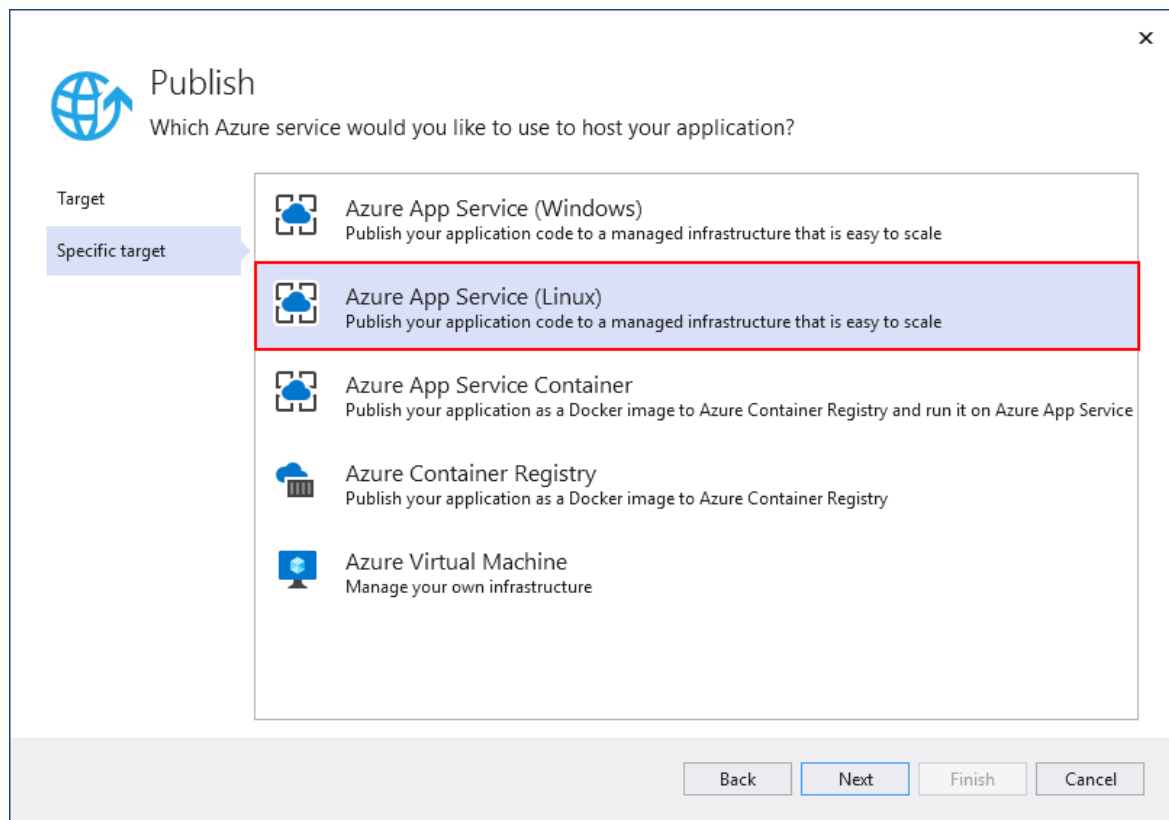


2. If you have previously configured any publishing profiles, the **Publish** window appears. Select **New**.
3. In the **Publish** window, select **Azure**.

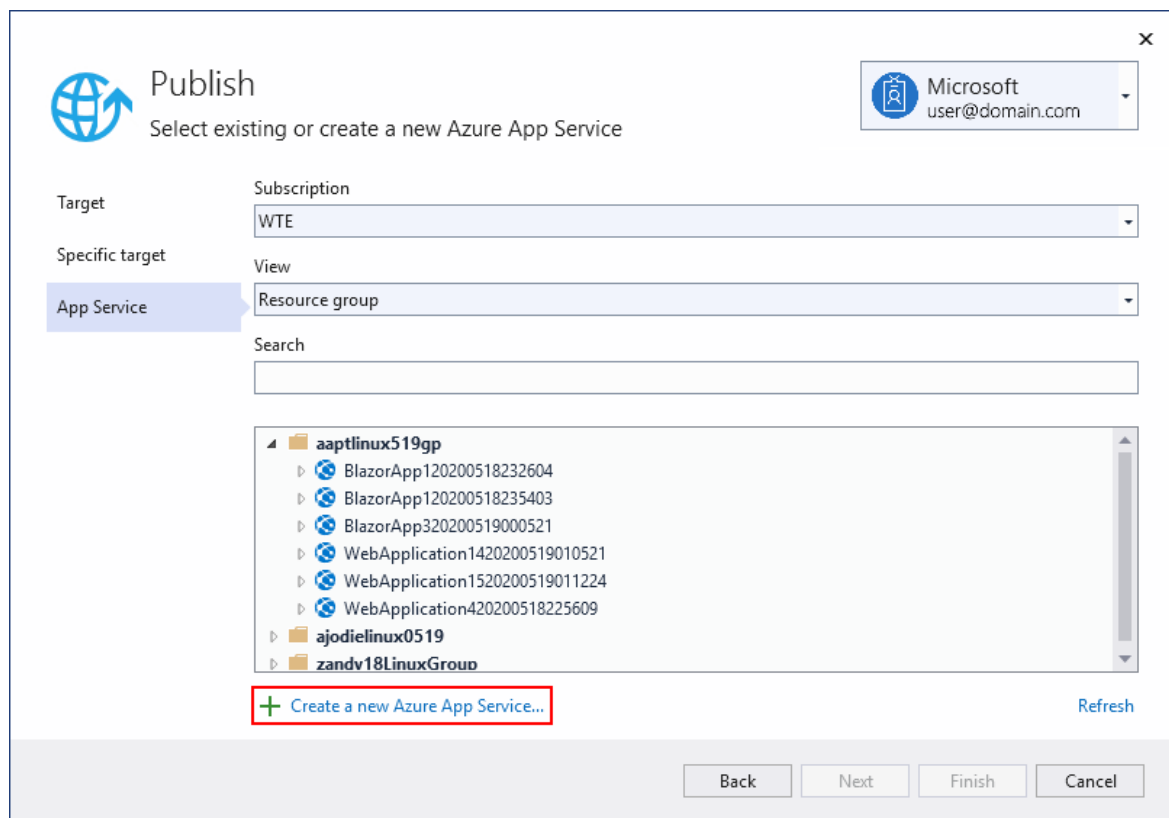


4. Select **Azure App Service (Linux)** and **Next**.





5. Sign in with you Azure account, if necessary. Select **Create a new Azure App Service...**



6. In the **Create Azure App Service (Linux)** dialog, the **App Name**, **Resource Group**, and **App Service Plan** entry fields are populated. You can keep these names or change them. When ready, select **Create**.

**App Service (Linux)**  
Create new

Microsoft  
user@domain.com

Name  
MyAspCoreWebAppOnAzure

Subscription  
WTE

Resource group  
MyAspCoreWebApp-rg\* [New...](#)

Hosting Plan  
MyAspCoreWebApp-hp\* (Central US, S1) [New...](#)

Export... Create Cancel

7. In the **Publish** dialog, the newly created instance has been automatically selected. When ready, click **Finish**.

**Publish**  
Select existing or create a new Azure App Service

Microsoft  
user@domain.com

Target  
Subscription  
WTE

Specific target  
View  
Resource group

App Service

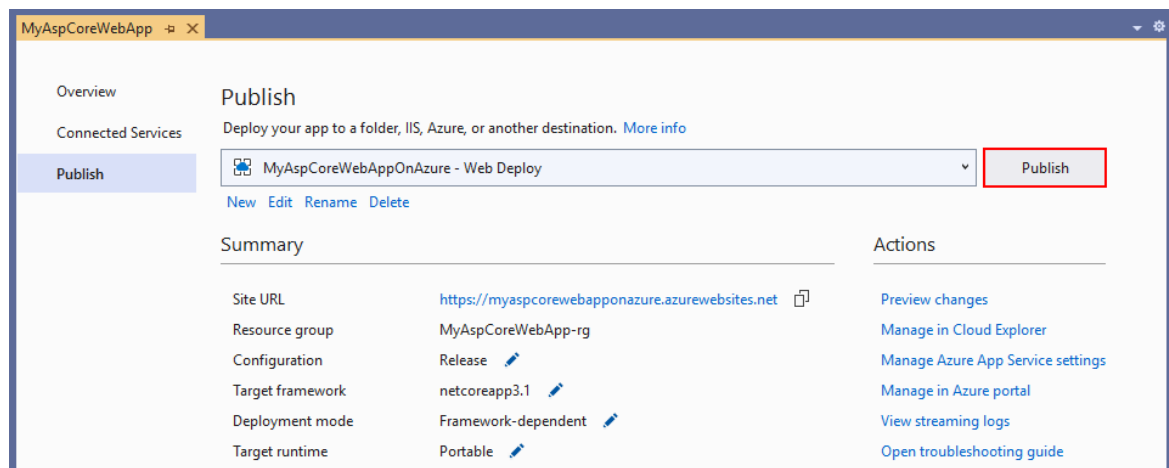
Search  
MyAspCoreWebAppOnAzure

MyAspCoreWebApp-rg  
 ▲ MyAspCoreWebAppOnAzure  
 ▸ Deployment Slots

+ Create a new Azure App Service... Refresh

Back Next Finish Cancel

8. Select **Publish**. Visual Studio deploys the app to your Azure App Service, and the web app loads in your browser. The project properties **Publish** pane shows the site URL and other details.



## Clean up resources

In the preceding steps, you created Azure resources in a resource group. If you don't expect to need these resources in the future, you can delete them by deleting the resource group. From the left menu in the Azure portal, select **Resource groups** and then select **myResourceGroup**. On the resource group page, make sure that the listed resources are the ones you want to delete. Select **Delete**, type **myResourceGroup** in the text box, and then select **Delete**.

## Next steps

In this quickstart, you learned how to use Visual Studio to create a publishing profile for deployment to App Service on Linux. You may want more information on publishing to Linux using Azure.

[Linux App Service](#)

# Publish a Web app to a web site using Visual Studio

3/5/2021 • 2 minutes to read • [Edit Online](#)

You can use the **Publish** tool to publish ASP.NET, ASP.NET Core, .NET Core, and Python apps to a website from Visual Studio. For Node.js, the steps are supported but the user interface is different.

## Prerequisites

- [Visual Studio 2019](#) installed with the appropriate workloads for your language of choice:
  - ASP.NET: **ASP.NET and web development**
  - Python: **Python development**
  - Node.js: **Node.js development**
- [Visual Studio 2017](#) installed with the appropriate workloads for your language of choice:
  - ASP.NET: **ASP.NET and web development**
  - Python: **Python development**
  - Node.js: **Node.js development**
- An ASP.NET, ASP.NET Core, Python, or Node.js project. If you don't already have a project, select an option below:
  - ASP.NET Core: Follow [Quickstart: Use Visual Studio to create your first ASP.NET Core web app](#), or use the following steps:

In Visual Studio 2019, choose **Create a new project** in the start window. If the start window is not open, choose **File > Start Window**. Type **web app** in the search box, choose **C#** as the language, then choose **ASP.NET Core Web Application (Model-View-Controller)**, and then choose **Next**. On the next screen, name the project **MyASPApp**, and then choose **Next**.

Choose either the recommended target framework (.NET Core 3.1) or .NET 5, and then choose **Create**.

In Visual Studio 2017, choose **File > New Project**, select **Visual C# > .NET Core**, then select **ASP.NET Core Web Application**. When prompted, select the **Web Application (Model-View-Controller)** template, make sure that **No Authentication** is selected, and then select **OK**.

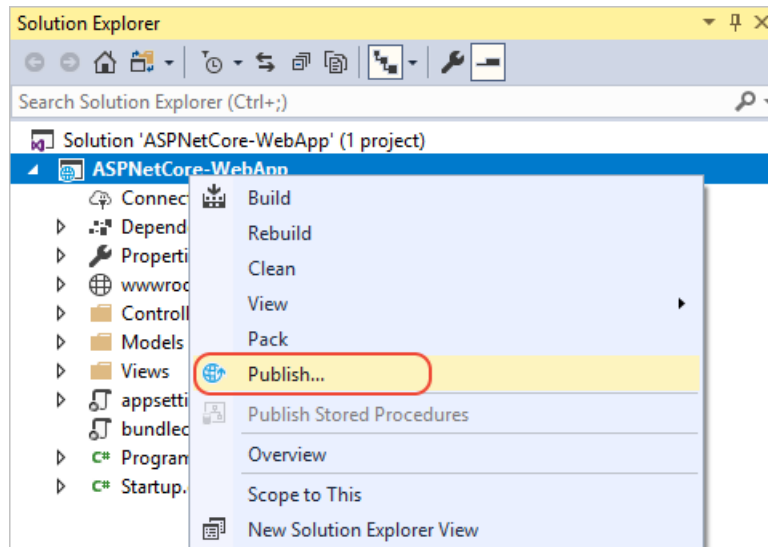
- Python: Follow [Quickstart: Create your first Python web app using Visual Studio](#), or use **File > New Project**, select **Python**, then select **Flask Web Project**.
  - Node.js: Follow [Quickstart: Use Visual Studio to create your first Node.js app](#), or use **File > New Project**, select **JavaScript**, then select **Blank Node.js Web Application**.
- Make sure you build the project using the **Build > Build Solution** menu command before following the deployment steps.

### NOTE

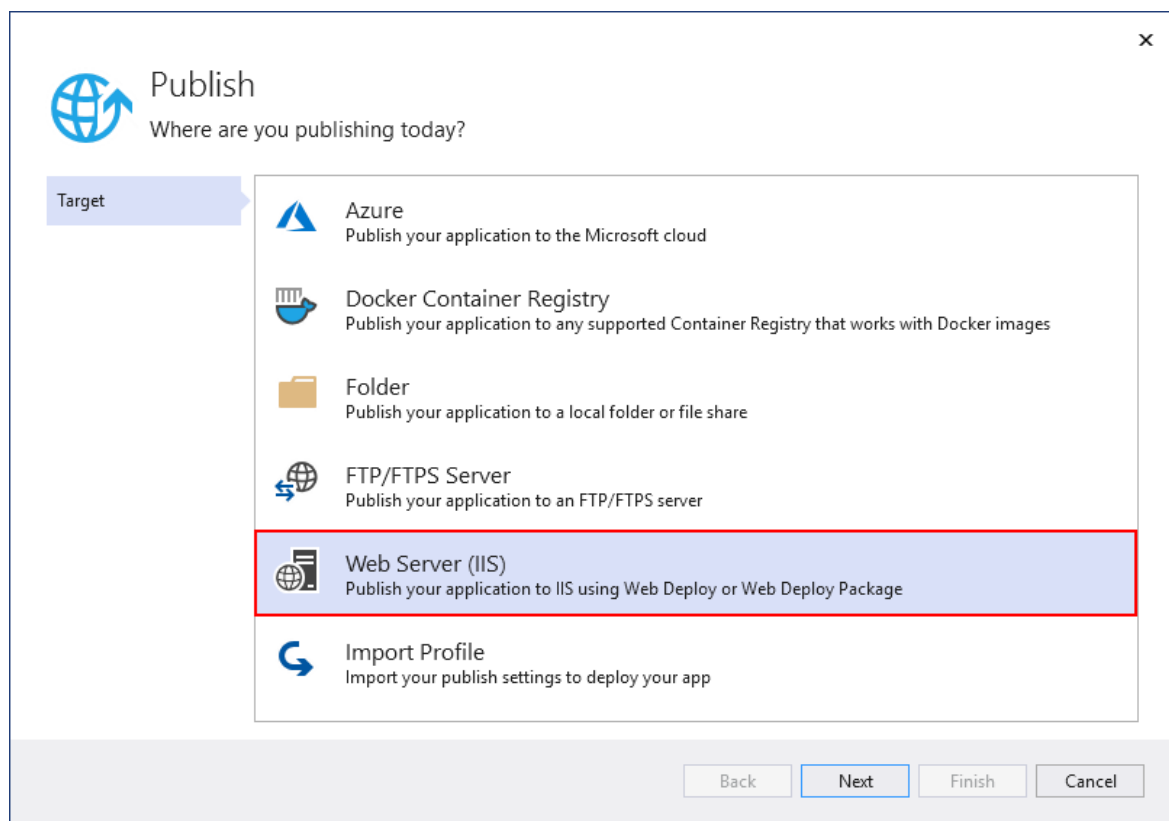
If you need to publish a Windows desktop application to a network file share, see [Deploy a desktop app using ClickOnce](#) (C# or Visual Basic). For C++/CLR, see [Deploy a native app using ClickOnce](#) or, for C/C++, see [Deploy a native app using a Setup project](#).

# Publish to a Web site

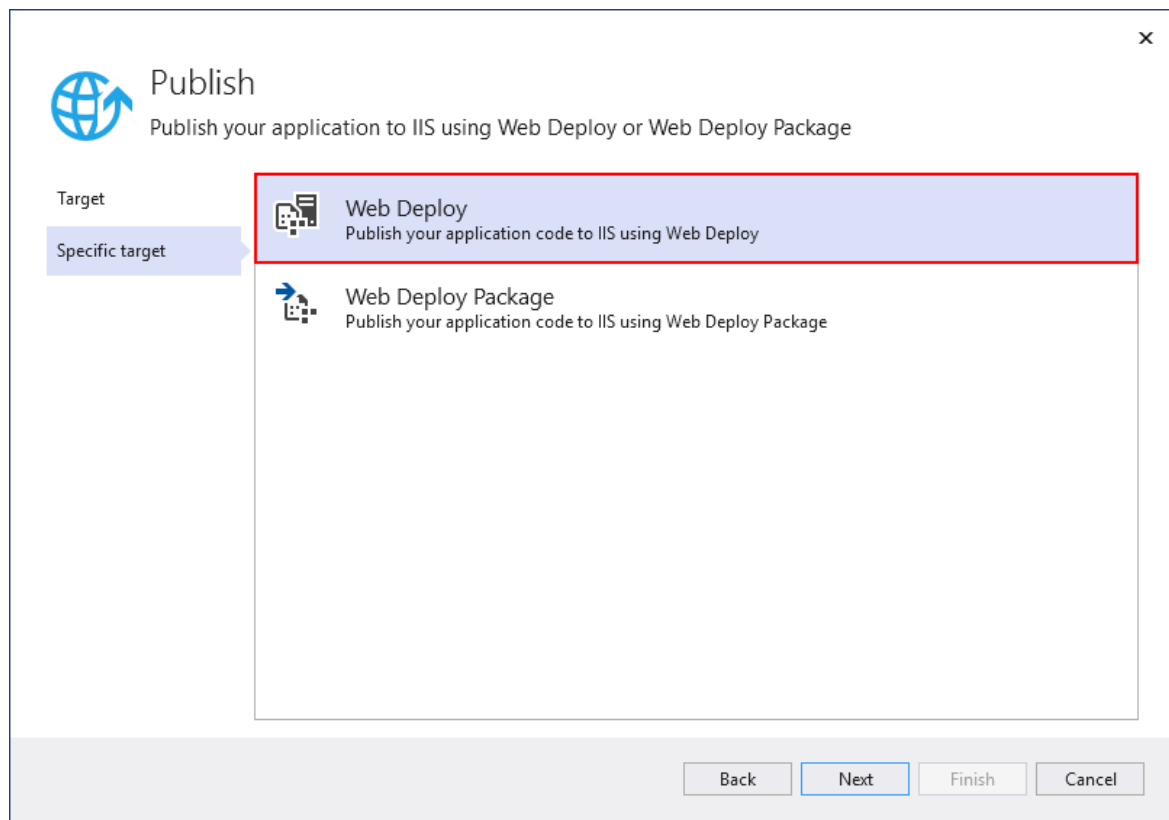
1. In Solution Explorer, right-click the project and choose **Publish** (or use the **Build** > **Publish** menu item).



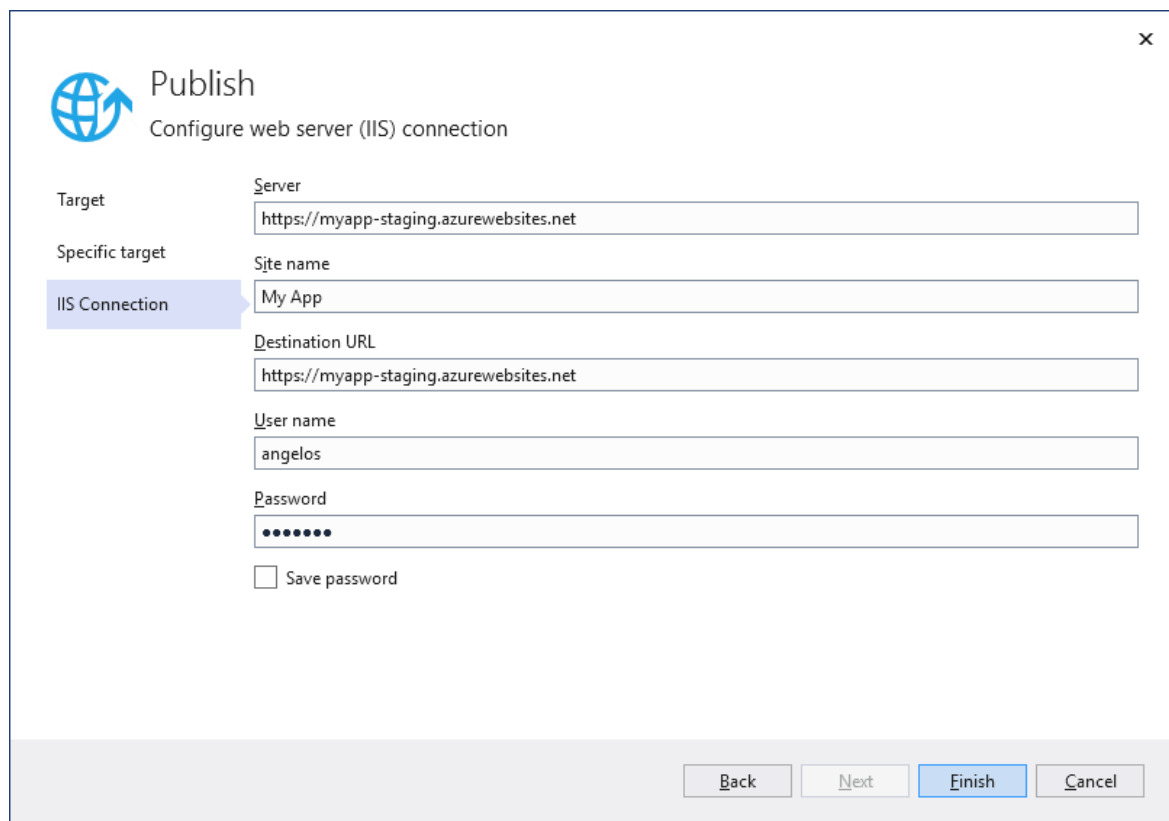
2. If you have previously configured any publishing profiles, the **Publish** pane appears. Select **New**.
3. In the **Publish** window, choose **Web Server (IIS)**.



4. Choose **Web Deploy** as the deployment method. Web Deploy simplifies deployment of Web applications and Web sites to IIS servers, and must be installed as an application on the server. Use the [Web platform installer](#) to install it.



5. Configure the required settings for the publish method and select **Finish**.



6. To publish, select **Publish** in the summary page. The Output window shows deployment progress and results.

If you need help troubleshooting ASP.NET Core on IIS, see [Troubleshoot ASP.NET Core on Azure App Service and IIS](#).

## Next steps

In this quickstart, you learned how to use Visual Studio to create a publishing profile. You can also configure a

publishing profile by importing publish settings.

[Import publish settings and deploy to IIS](#)

# Deploy an app to a folder using Visual Studio

3/6/2021 • 3 minutes to read • [Edit Online](#)

You can use the **Publish** tool to publish ASP.NET, ASP.NET Core, .NET Core, and Python apps to a folder from Visual Studio. For Node.js, the steps are supported but the user interface is different.

## Prerequisites

- [Visual Studio 2019](#) installed with the appropriate workloads for your language of choice:
  - ASP.NET: **ASP.NET and web development**
  - Python: **Python development**
  - Node.js: **Node.js development**
- [Visual Studio 2017](#) installed with the appropriate workloads for your language of choice:
  - ASP.NET: **ASP.NET and web development**
  - Python: **Python development**
  - Node.js: **Node.js development**
- An ASP.NET, ASP.NET Core, Python, or Node.js project. If you don't already have a project, select an option below:

- ASP.NET Core: Follow [Quickstart: Use Visual Studio to create your first ASP.NET Core web app](#), or use the following steps:

In Visual Studio 2019, choose **Create a new project** in the start window. If the start window is not open, choose **File > Start Window**. Type **web app** in the search box, choose **C#** as the language, then choose **ASP.NET Core Web Application (Model-View-Controller)**, and then choose **Next**. On the next screen, name the project **MyASPApp**, and then choose **Next**.

Choose either the recommended target framework (.NET Core 3.1) or .NET 5, and then choose **Create**.

In Visual Studio 2017, choose **File > New Project**, select **Visual C# > .NET Core**, then select **ASP.NET Core Web Application**. When prompted, select the **Web Application (Model-View-Controller)** template, make sure that **No Authentication** is selected, and then select **OK**.

- Python: Follow [Quickstart: Create your first Python web app using Visual Studio](#), or use **File > New Project**, select **Python**, then select **Flask Web Project**.
  - Node.js: Follow [Quickstart: Use Visual Studio to create your first Node.js app](#), or use **File > New Project**, select **JavaScript**, then select **Blank Node.js Web Application**.
- Make sure you build the project using the **Build > Build Solution** menu command before following the deployment steps.

### NOTE

If you need to publish a Windows desktop application to a folder, see [Deploy a desktop app using ClickOnce](#) (C# or Visual Basic). For C++/CLR, see [Deploy a native app using ClickOnce](#) or, for C/C++, see [Deploy a native app using a Setup project](#).

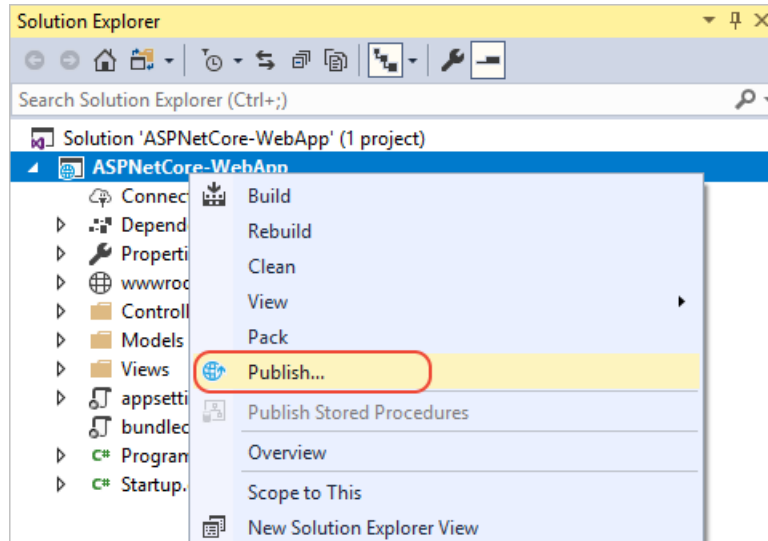


## NOTE

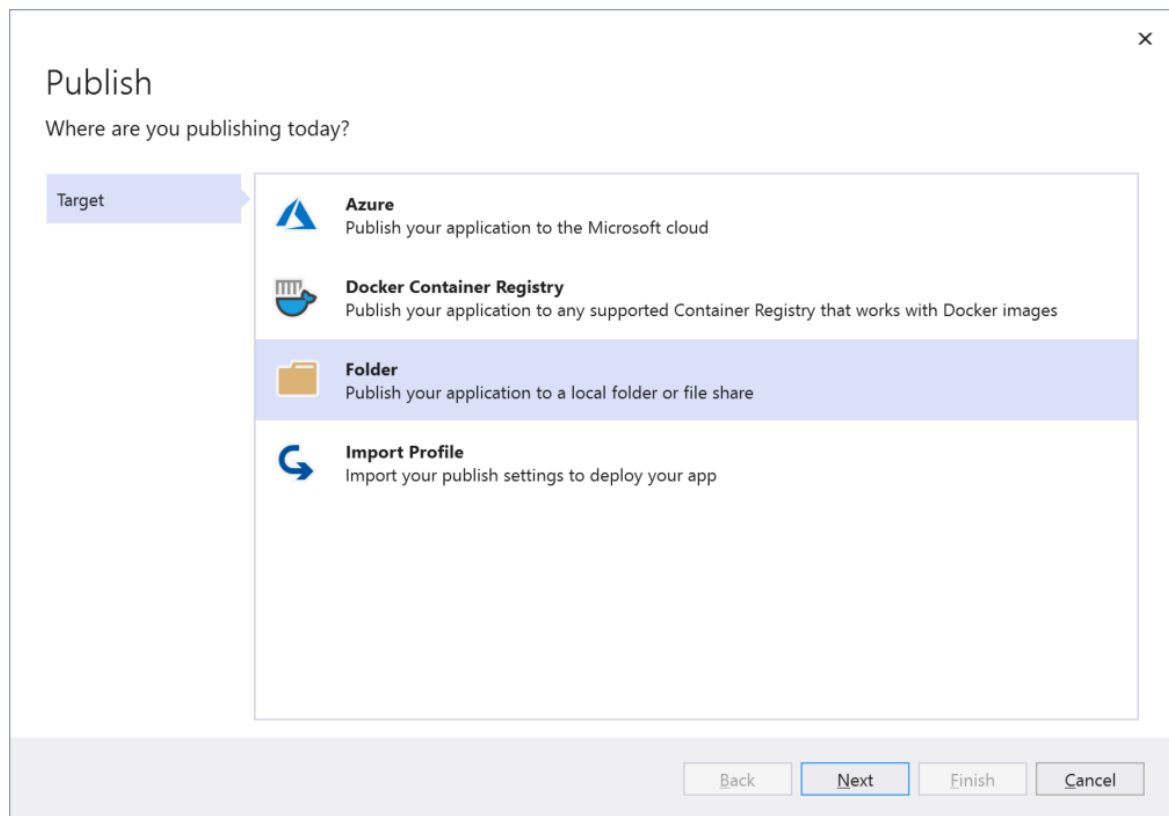
If you need to publish a .NET Core 3.1, or newer, Windows desktop application to a folder, see [Deploy a .NET Windows application using ClickOnce](#).

# Deploy to a local folder

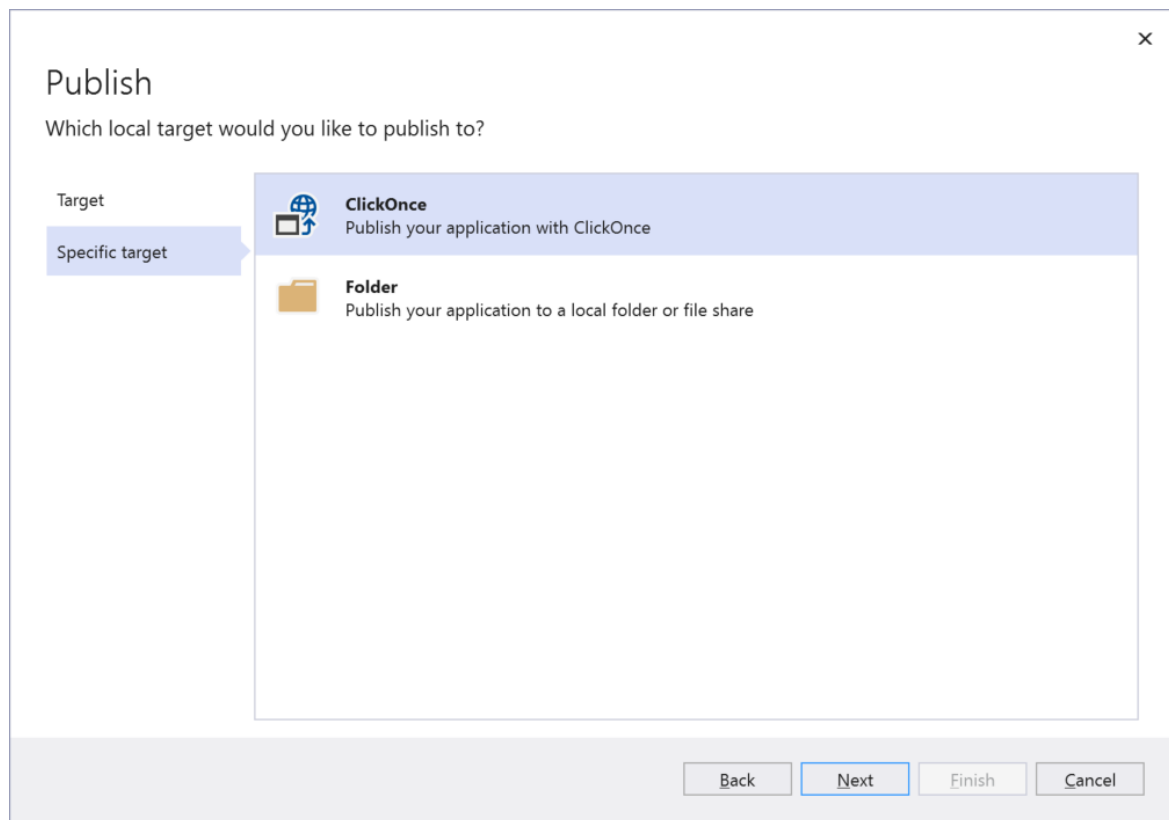
1. In Solution Explorer, right-click the project and choose **Publish** (or use the **Build** > **Publish** menu item).



2. If you have previously configured any publishing profiles, the **Publish** window appears. Select **New**.
3. In the **Publish** window, select **Folder**.

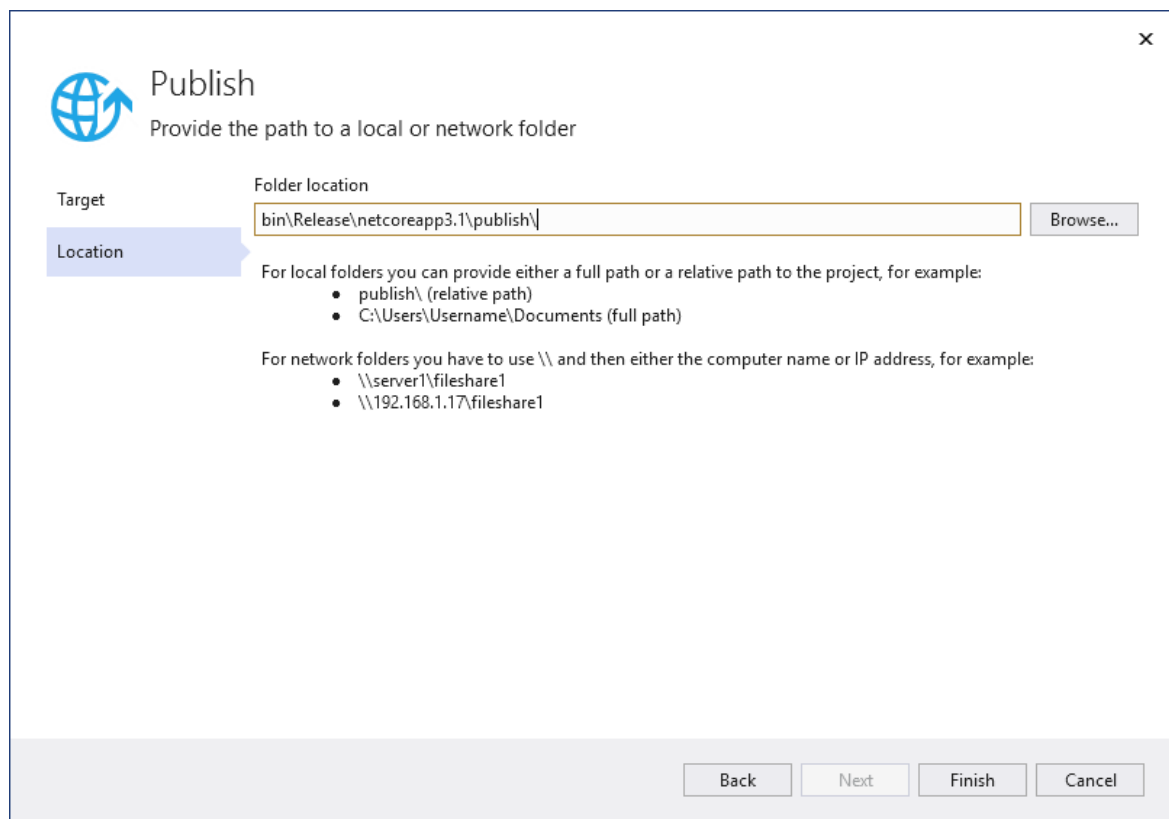


If you are deploying a .NET Core 3.1, or newer, Windows Application you may need to select **Folder** in the **Specific target** window.

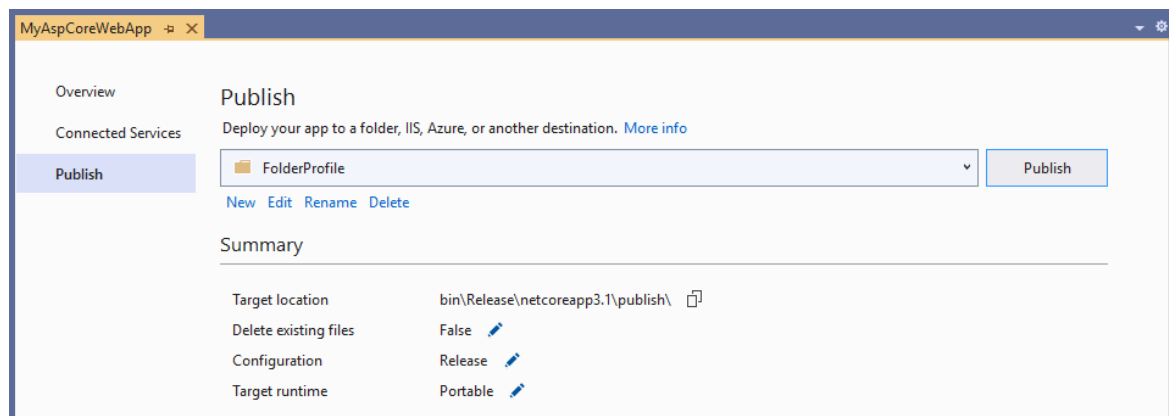


If you wish to publish a .NET Core 3.1, or newer, Windows application with ClickOnce, see [Deploy a .NET Windows application using ClickOnce](#).

4. Enter a path or select **Browse** to specify a folder.

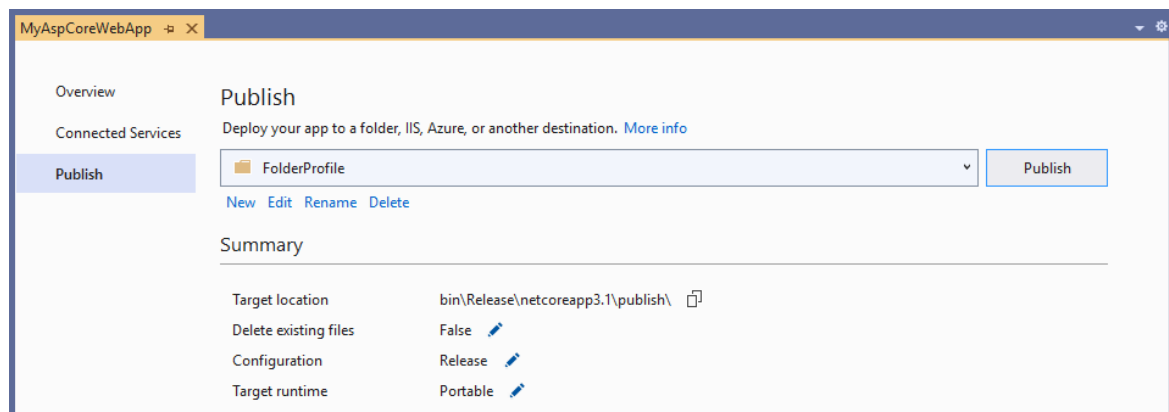


Click **Finish** to save the profile.



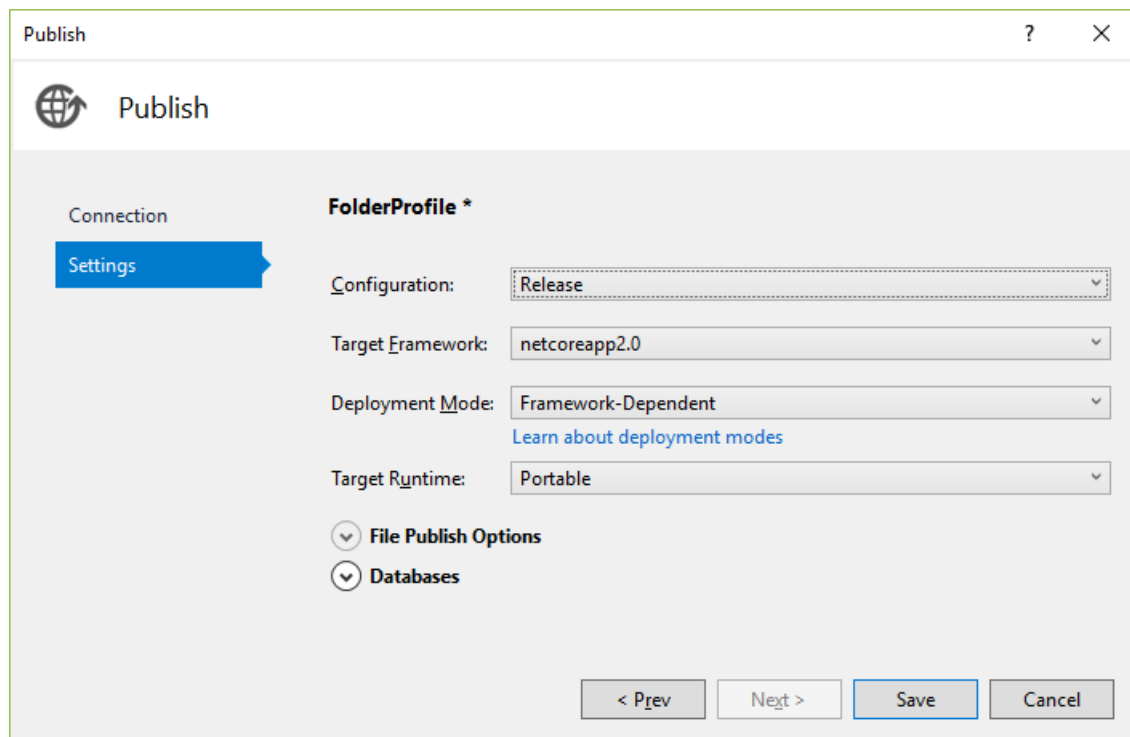
5. Select **Publish**. Visual Studio builds the project and publishes it to the specified folder.

The project properties **Publish** pane appears, showing a profile summary.



6. To configure deployment settings, select **Edit** in the publish profile summary and select the **Settings** tab.

The settings you see depend on your application type. The following illustration shows example settings for an ASP.NET Core app.



For additional help to choose settings in .NET, see the following:

- [Framework-dependent vs. self-contained deployment](#)
- [Target runtime identifiers \(portable RID, et al\)](#)

- [Debug and release configurations](#)

7. Configure options such as whether to deploy a Debug or Release configuration, and then select **Save**.

8. To republish, select **Publish**.

Deploy the published files in any way you like. For example, you can package them in a *.zip* file, use a simple copy command, or deploy them with any installation package of your choice.

## Next steps

For .NET apps:

- [Deploy a .NET Core Application with the Publish tool](#)
- [.NET Core application publishing \(framework-dependent vs. self-contained deployments\)](#)
- [Deploy the .NET Framework and applications](#)
- [Deploy a .NET Windows application using ClickOnce.](#)

# Deploy a .NET Windows desktop application using ClickOnce

3/5/2021 • 2 minutes to read • [Edit Online](#)

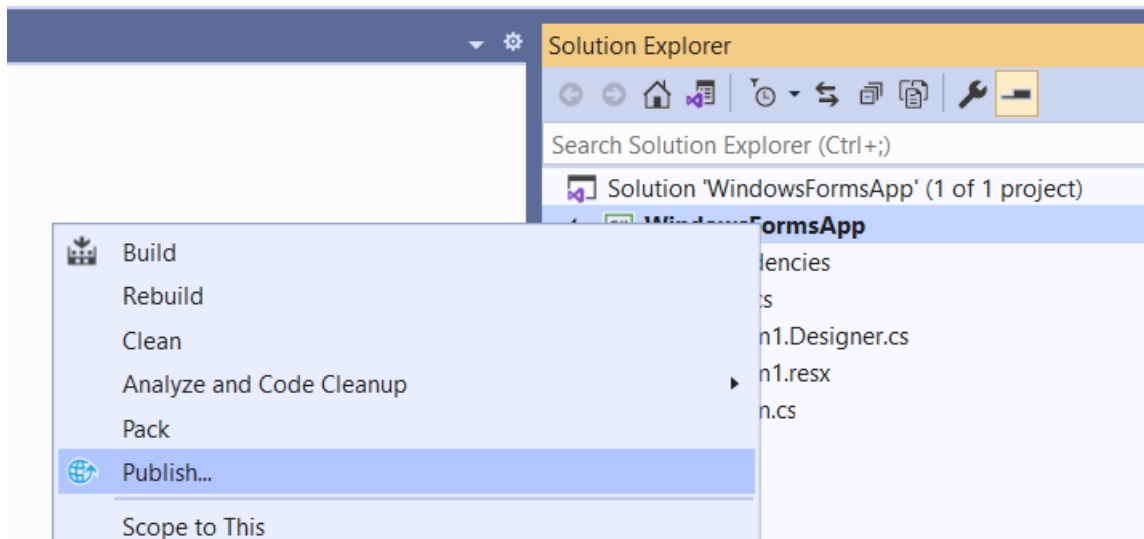
Starting in Visual Studio 2019 version 16.8, you can use the **Publish** tool to publish .NET Core 3.1, or newer, Windows Desktop applications using ClickOnce from Visual Studio.

## NOTE

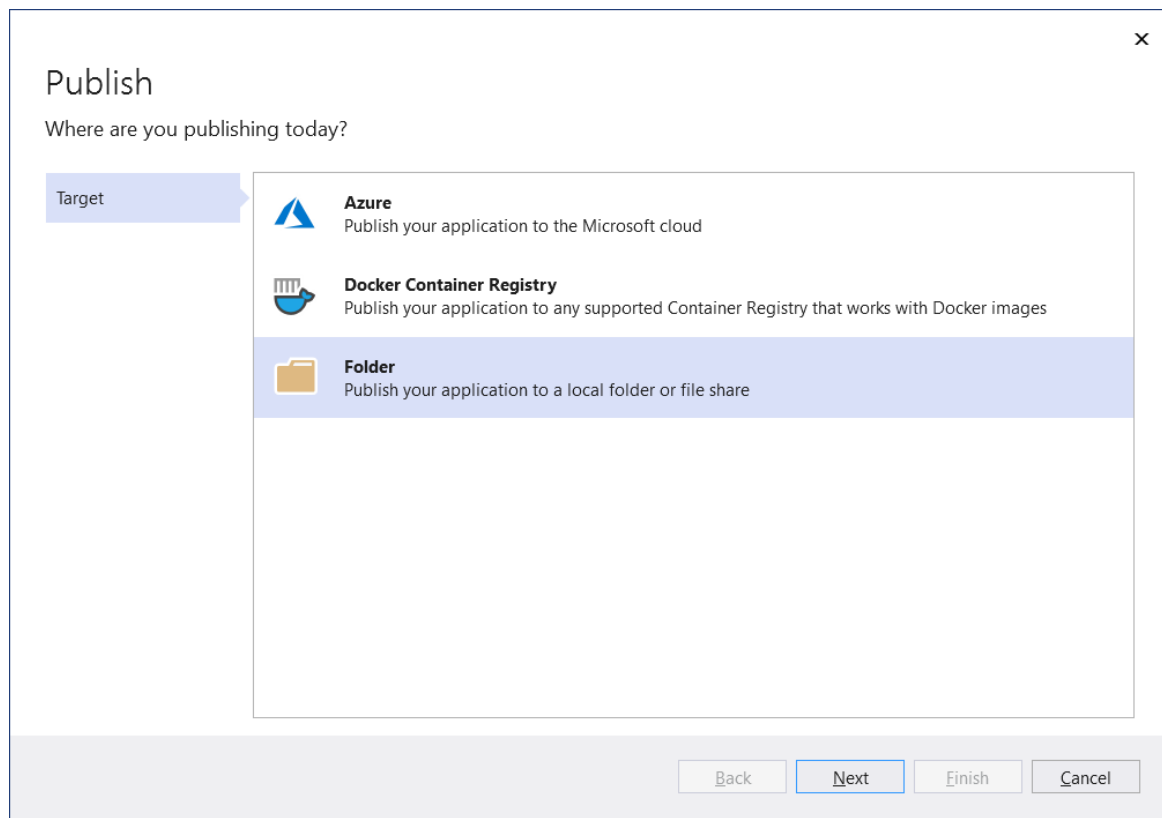
If you need to publish a .NET Framework Windows application, see [Deploy a desktop app using ClickOnce](#) (C# or Visual Basic).

## Publishing with ClickOnce

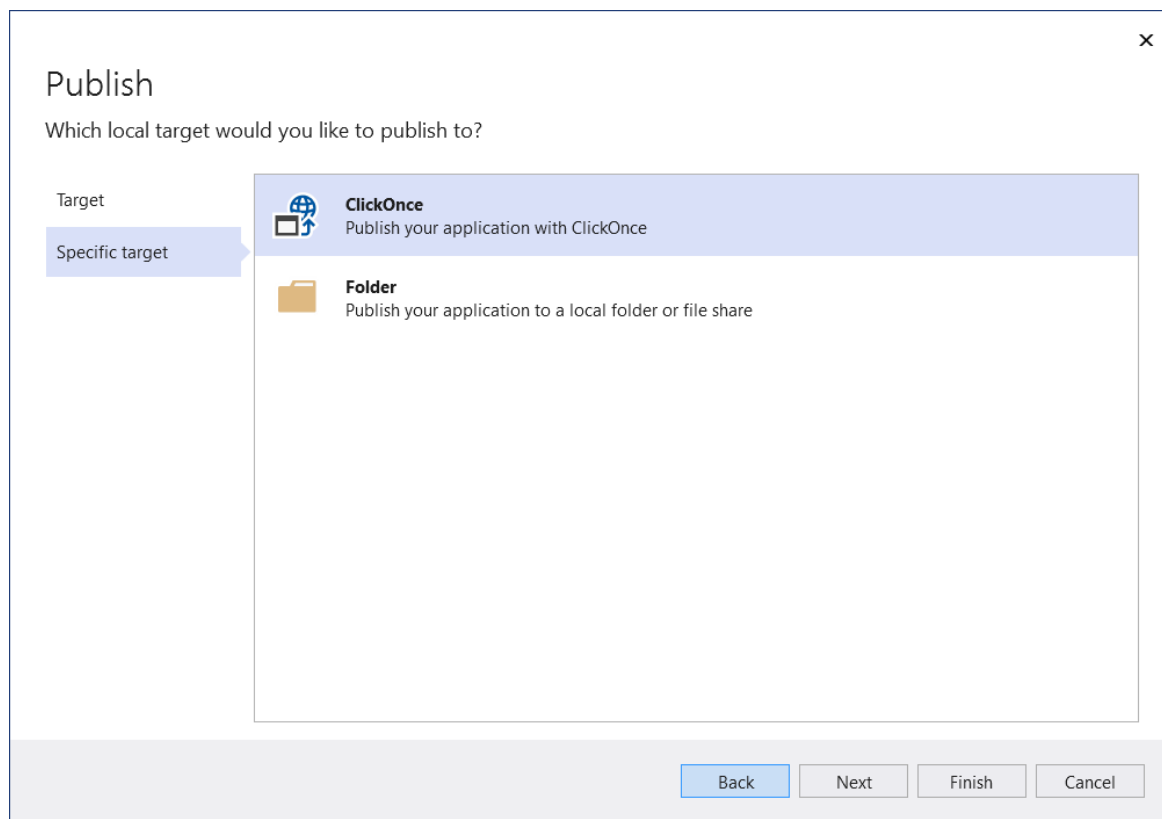
1. In Solution Explorer, right-click the project and choose **Publish** (or use the **Build** > **Publish** menu item).



2. If you have previously configured any publishing profiles, the **Publish** page appears. Select **New**.
3. In the **Publish** wizard, select **Folder**.



4. In the **Specific target** page, select **ClickOnce**.



5. Enter a path or select **Browse** to select the publish location.

**Publish** [Close]

Specify the path to a local folder or file share

Target

Specific target

**Publish location**

Install location

Settings

Sign manifests

Configuration

Publish location: bin\publish\ [Browse...]

Examples

- Local folder: C:\deploy\myapplication
- File share: \\server\myapplication

[Back] [Next] [Finish] [Cancel]

6. In the **Install location** page, select where users will install the application from.

**Publish** [Close]

How will users install the application?

Target

Specific target

Publish location

**Install location**

Settings

Sign manifests

Configuration

☐ From a web site  
 Specify the URL

☐ From a UNC path or file share  
 Specify the UNC path  
 [Browse...]

☒ From a CD, DVD, or USB drive

[Back] [Next] [Finish] [Cancel]

7. In the **Settings** page, you can provide the settings necessary for ClickOnce.

8. If you selected to install from a UNC path or web site, this page allows you to specify whether the application is available offline. When selected, this option will list the application on the users Start Menu and it allows the application to be automatically updated when a new version is published. By default, updates are available from the Install location. If you wish to have a different location for updates, you can specify that using the Update Settings link. If you do not want the application to be available offline, it will run from the install location.





## NOTE

The Publish version number is unique for each ClickOnce profile. If you plan on having more than one profile, you will need to keep this in mind.

10. In the **Sign manifests** page, you can specify if the manifests should be signed and which certificate to use.

The screenshot shows the 'Publish' dialog box with the 'Sign manifests' page selected. The page title is 'Publish' and the subtitle is 'Select options to sign the ClickOnce manifest'. On the left, a sidebar lists the steps: Target, Specific target, Publish location, Install location, Settings, Sign manifests (highlighted), and Configuration. The main area contains the following options:

- Target:** A checkbox labeled 'Sign the ClickOnce manifests' is checked.
- Specific target:** Three links are available: 'Select from store', 'Select from file', and 'Create test certificate'.
- Publish location:** A section header.
- Install location:** A section header.
- Certificate selected:** A dropdown menu showing '<Required>'.
- Issued to:** (none)
- Issued by:** (none)
- Intended purpose:** (none)
- Expiration date:** (none)
- Signature algorithm:** (none)
- Details...** (link)
- Timestamp server URL:** A text input field.

At the bottom right, there are four buttons: 'Back', 'Next', 'Finish', and 'Cancel'.

11. On the **Configuration** page, you can select the desired project configuration.

The screenshot shows the 'Publish' dialog box with the 'Configuration' page selected. The page title is 'Publish' and the subtitle is 'Specify project configuration'. On the left, a sidebar lists the steps: Target, Specific target, Publish location, Install location, Settings, Sign manifests, and Configuration (highlighted). The main area contains the following options:

- Target:** A section header.
- Configuration:** A dropdown menu showing 'Release | Any CPU'.
- Specific target:** A section header.
- Target framework:** A dropdown menu showing 'net5.0'.
- Publish location:** A section header.
- Deployment mode:** A dropdown menu showing 'Framework-dependent'.
- Install location:** A section header.
- Target runtime:** A dropdown menu showing 'Portable'.

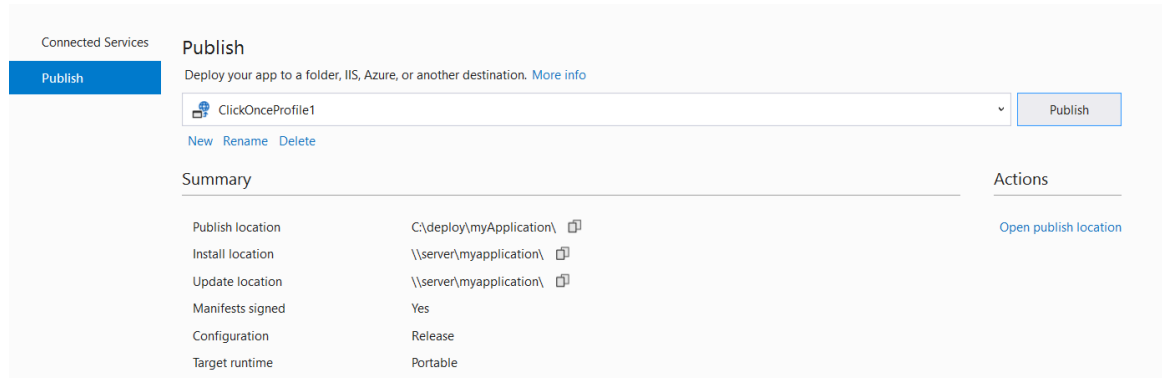
At the bottom right, there are four buttons: 'Back', 'Next', 'Finish', and 'Cancel'.

For additional help on which setting to choose, see the following:

- [Framework-dependent vs. self-contained deployment](#)
- [Target runtime identifiers \(portable RID, et al\)](#)
- [Debug and release configurations](#)

12. Select **Finish** to save the new ClickOnce Publish Profile.

13. On the **Summary** page, select **Publish** and Visual Studio builds the project and publishes it to the specified publish folder. This page also shows a profile summary.



14. To republish, select **Publish**.

## Next steps

For .NET apps:

- [Deploy the .NET Framework and applications](#)
- [ClickOnce reference](#)

# How to: Publish a ClickOnce application using the Publish Wizard

3/5/2021 • 3 minutes to read • [Edit Online](#)

To make a ClickOnce application available to users, you must publish it to a file share or path, FTP server, or removable media. You can publish the application by using the Publish Wizard; additional properties related to publishing are available on the **Publish** page of the **Project Designer**. For more information, see [Publishing ClickOnce applications](#).

Before you run the Publish Wizard, you should set the publishing properties appropriately. For example, if you want to designate a key to sign your ClickOnce application, you can do so on the **Signing** page of the **Project Designer**. For more information, see [Secure ClickOnce applications](#).

## NOTE

When you install more than one version of an application by using ClickOnce, the installation moves earlier versions of the application into a folder named *Archive*, in the publish location that you specify. Archiving earlier versions in this manner keeps the installation directory clear of folders from the earlier version.

## NOTE

The dialog boxes and menu commands you see might differ from those described in Help, depending on your active settings or edition. To change your settings, click **Import and Export Settings** on the **Tools** menu. For more information, see [Reset settings](#).

## To publish to a file share or path

1. In **Solution Explorer**, select the application project.
2. On the **Build** menu, click **Publish *Projectname***.  
  
The Publish Wizard appears.
3. In the **Where do you want to publish the application?** page, enter a valid FTP server address or a valid file path using one of the formats shown, and then click **Next**.
4. In the **How will users install the application?** page, select the location where users will go to install the application:
  - If users will install from a Web site, click **From a Web site** and enter a URL that corresponds to the file path entered in the previous step. Click **Next**. (This option is typically used when you specify an FTP address as the publishing location. Direct download from FTP is not supported. Therefore, you have to enter a URL here.)
  - If users will install the application directly from the file share, click **From a UNC path or file share**, and then click **Next**. (This is for publishing locations of the form `c:\deploy\myapp` or `\\server\myapp`.)
  - If users will install from removable media, click **From a CD-ROM or DVD-ROM**, and then click **Next**.

5. On the **Will the application be available offline?** page, click the appropriate option:

- If you want to enable the application to be run when the user is disconnected from the network, click **Yes, this application will be available online or offline**. A shortcut on the **Start** menu will be created for the application.
- If you want to run the application directly from the publish location, click **No, this application is only available online**. A shortcut on the **Start** menu will not be created.

Click **Next** to continue.

6. Click **Finish** to publish the application.

Publishing status is displayed in the status notification area.

## To publish to a CD-ROM or DVD-ROM

1. In **Solution Explorer**, right-click the application project and click **Properties**.

The **Project Designer** appears.

2. Click the **Publish** tab to open the **Publish** page in the **Project Designer**, and click the **Publish Wizard** button.

The Publish Wizard appears.

3. In the **Where do you want to publish the application?** page, enter the file path or FTP location where the application will be published, for example *d:\deploy*. Then click **Next** to continue.

4. On the **How will users install the application?** page, click **From a CD-ROM or DVD-ROM**, and then click **Next**.

### NOTE

If you want the installation to run automatically when the CD-ROM is inserted into the drive, open the **Publish** page in the **Project Designer** and click the **Options** button, and then, in the **Publish Options** wizard, select **For CD installations, automatically start Setup when CD is inserted**.

5. If you distribute your application on CD-ROM, you might want to provide updates from a Web site. In the **Where will the application check for updates?** page, choose an update option:

- If the application will check for updates, click **The application will check for updates from the following location** and enter the location where updates will be posted. This can be a file location, Web site, or FTP server.
- If the application will not check for updates, click **The application will not check for updates**.

Click **Next** to continue.

6. Click **Finish** to publish the application.

Publishing status is displayed in the status notification area.

### NOTE

After publishing is complete, you will have to use a CD-Rewriter or DVD-Rewriter to copy the files from the location specified in step 3 to the CD-ROM or DVD-ROM media.

## See also

- [ClickOnce security and deployment](#)
- [Secure ClickOnce applications](#)
- [Deploying an Office solution by using ClickOnce](#)

# First look at deployment in Visual Studio

3/5/2021 • 5 minutes to read • [Edit Online](#)

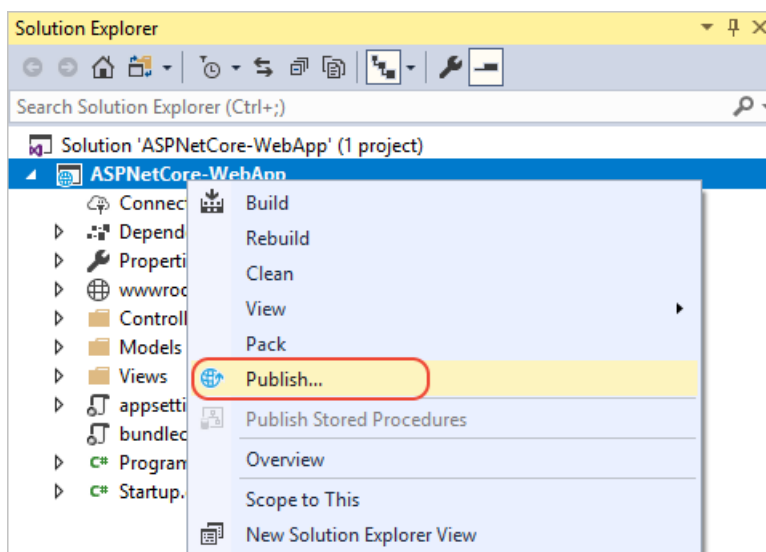
By deploying an application, service, or component, you distribute it for installation on other computers, devices, or servers, or in the cloud. You choose the appropriate method in Visual Studio for the type of deployment that you need. (Many app types support other deployment tools, such as command-line deployment or NuGet, that aren't described here.)

See the quickstarts and tutorials for step-by-step deployment instructions. For an overview of deployment options, see [What publishing options are right for me?](#).

## Deploy to a local folder

Deployment to a local folder is typically used for testing or to begin a staged deployment in which another tool is used for final deployment.

- **ASP.NET, ASP.NET Core, Node.js, Python, and .NET Core:** Use the **Publish** tool to deploy to a local folder. The exact options available depend on your app type. In Solution Explorer, right-click your project and select **Publish**. (If you haven't previously configured any publishing profiles, you must then select **Create new profile**.) Next, select **Folder**. For more information, see [Deploy to a local folder](#).



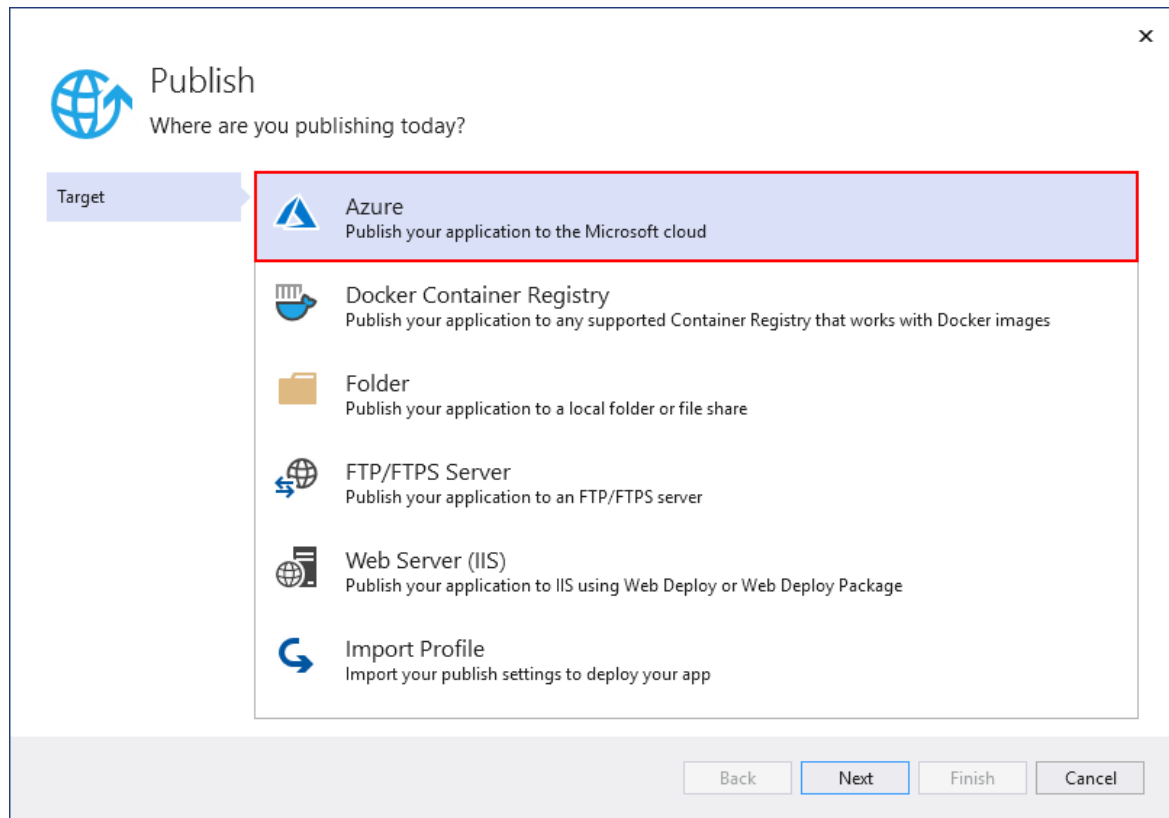
- **Windows desktop:** You can publish a Windows desktop application to a folder by using ClickOnce deployment. Users can then install the application with a single click. For more information, see the following articles:
  - [Deploy a .NET Framework Windows desktop app using ClickOnce](#).
  - [Deploy a .NET Windows desktop app using ClickOnce](#).
  - [Deploy a C++/CLR app using ClickOnce](#) or, for C/C++, see [Deploy a native app using a Setup project](#).

## Publish to Azure

- **ASP.NET, ASP.NET Core, Python, and Node.js:** Publish to Azure App Service or Azure App Service on Linux (using containers) by using one of the following methods:
  - For continuous (or automated) deployment of apps, use Azure DevOps with [Azure Pipelines](#).
  - For one-time (or manual) deployment of apps, use the **Publish** tool in Visual Studio.

For deployment that provides more customized configuration of the server, you can also use the **Publish** tool to deploy apps to an Azure virtual machine.

To use the **Publish** tool, right-click the project in Solution Explorer and select **Publish**. (If you've previously configured any publishing profiles, you must then select **Create new profile**.) In the **Publish** dialog box, select either **App Service** or **Azure Virtual Machines**, and then follow the configuration steps.



Starting in Visual Studio 2017 version 15.7, you can deploy ASP.NET Core apps to App Service on Linux.

For Python apps, also see [Python - Publishing to Azure App Service](#).

For a quick introduction, see [Publish to Azure](#) and [Publish to Linux](#). Also, see [Publish an ASP.NET Core app to Azure](#). For deployment using Git, see [Continuous deployment of ASP.NET Core to Azure with Git](#).

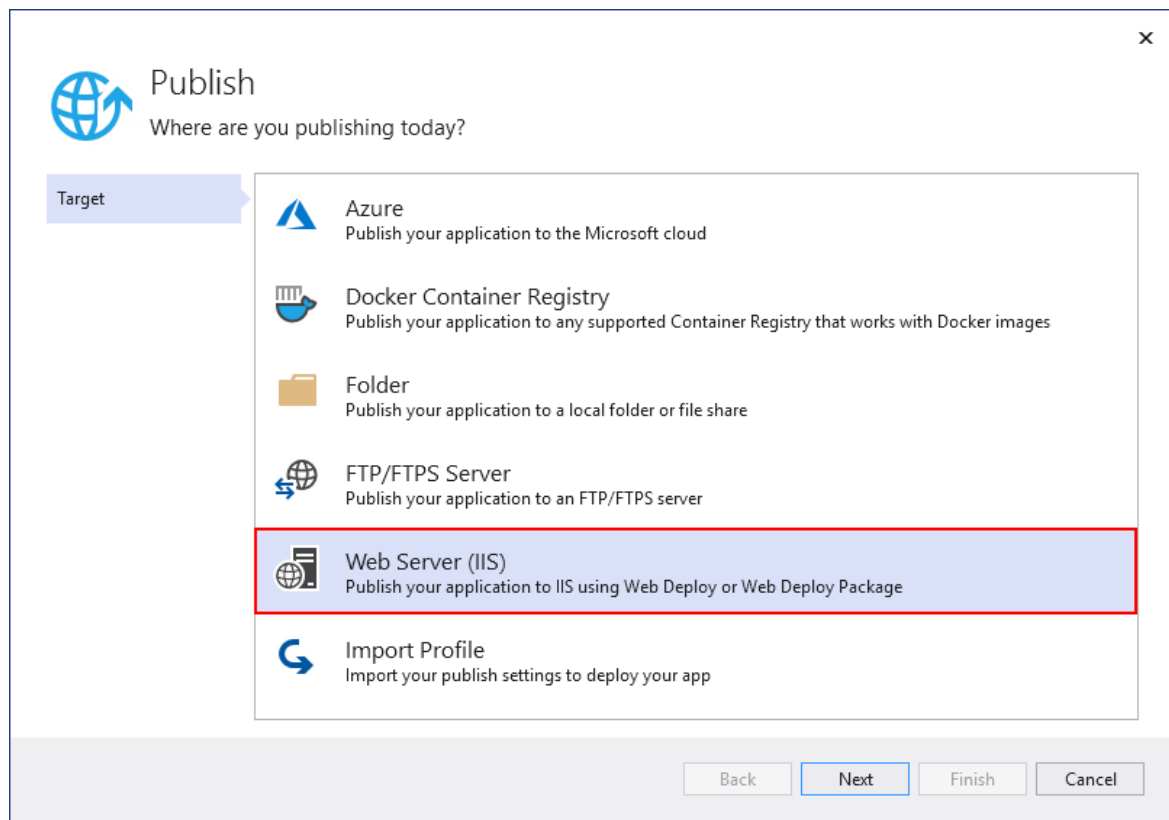
#### NOTE

If you don't already have an Azure account, you can [sign up here](#).

## Publish to the web or deploy to a network share

- **ASP.NET, ASP.NET Core, Node.js, and Python:** You can use the **Publish** tool to deploy to a website by using FTP or Web Deploy. For more information, see [Deploy to a website](#).

In Solution Explorer, right-click the project and select **Publish**. (If you've previously configured any publishing profiles, you must then select **Create new profile**.) In the **Publish** tool, select the option you want and follow the configuration steps.



For information on importing a publish profile in Visual Studio, see [Import publish settings and deploy to IIS](#).

You can also deploy ASP.NET applications and services in a number of other ways. For more information, see [Deploying ASP.NET web applications and services](#).

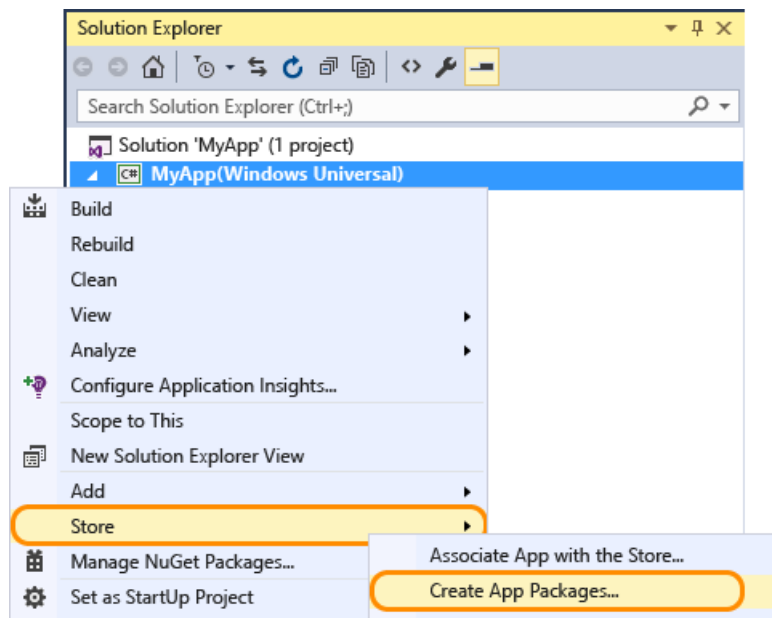
- **Windows desktop:** You can publish a Windows desktop application to a web server or a network file share by using ClickOnce deployment. Users can then install the application with a single click. For more information, see the following articles:
  - [Deploy a .NET Framework Windows desktop app using ClickOnce](#)
  - [Deploy a .NET Windows desktop app using ClickOnce](#)
  - [Deploy a C++/CLR app using ClickOnce](#)

## Publish to Microsoft Store

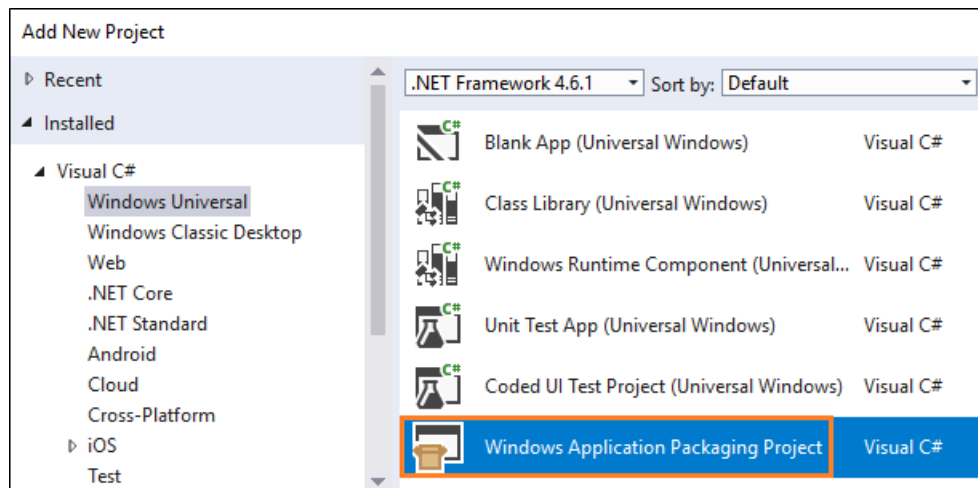
From Visual Studio, you can create app packages for deployment to Microsoft Store.

- **UWP:** You can package your app and deploy it by using menu items. For more information, see [Package a UWP app by using Visual Studio](#).





- **Windows desktop:** You can deploy to Microsoft Store by using the Desktop Bridge starting in Visual Studio 2017 version 15.4. To do this, start by creating a Windows Application Packaging Project. For more information, see [Package a desktop app for Microsoft Store \(Desktop Bridge\)](#).



## Deploy to a device (UWP)

If you're deploying a UWP app for testing on a device, see [Run UWP apps on a remote machine in Visual Studio](#).

## Create an installer package (Windows desktop)

If you require a more complex installation of a desktop application than ClickOnce can provide, you can create a Windows Installer package (MSI or EXE installation file) or a custom bootstrapper.

- An MSI-based installer package can be created by using the [WiX Toolset Visual Studio 2017 Extension](#). This is a command-line toolset.
- An MSI or EXE installer package can be created by using [InstallShield](#) from Flexera Software. InstallShield may be used with Visual Studio 2017 and later versions. Community Edition isn't supported.

### NOTE

InstallShield Limited Edition is no longer included with Visual Studio and isn't supported in Visual Studio 2017 and later versions. Check with [Flexera Software](#) about future availability.

- An MSI or EXE installer package can be created by using a Setup project (vdproj). To use this option, install the [Visual Studio Installer Projects extension](#).
- You can also install prerequisite components for desktop applications by configuring a generic installer, which is known as a bootstrapper. For more information, see [Application deployment prerequisites](#).

## Deploy to a test lab

You can enable more sophisticated development and testing by deploying your applications into virtual environments. For more information, see [Test on a lab environment](#).

## Continuous deployment

You can use Azure Pipelines to enable continuous deployment of your app. For more information, see [Azure Pipelines](#) and [Deploy to Azure](#).

## Deploy a SQL database

- [Change target platform and publish a database project \(SQL Server Data Tools \(SSDT\)\)](#)
- [Deploy an Analysis Services Project \(SSAS\)](#)
- [Deploy Integration Services \(SSIS\) projects and packages](#)
- [Build and deploy to a local database](#)

## Deployment for other app types

| APP TYPE             | DEPLOYMENT SCENARIO   | LINK   |
|----------------------|---|--|
| Office app           | You can publish an add-in for Office from Visual Studio.  | <a href="#">Deploy and publish your Office add-in</a>      |
| WCF or OData service | Other applications can use WCF RIA services that you deploy to a web server.  | <a href="#">Developing and deploying WCF Data Services</a> |
| LightSwitch          | LightSwitch is no longer supported starting in Visual Studio 2017, but can still be deployed from Visual Studio 2015 and earlier. | <a href="#">Deploying LightSwitch applications</a>         |

## Next steps

In this tutorial, you took a quick look at deployment options for different applications.

[What publishing options are right for me?](#)

# Build ClickOnce applications from the command line

3/5/2021 • 8 minutes to read • [Edit Online](#)

In Visual Studio, you can build projects from the command line, even if they are created in the integrated development environment (IDE). In fact, you can rebuild a project created with Visual Studio on another computer that has only the .NET Framework installed. This allows you to reproduce a build using an automated process, for example, in a central build lab or using advanced scripting techniques beyond the scope of building the project itself.

## Use MSBuild to reproduce .NET Framework ClickOnce application deployments

When you invoke `msbuild /target:publish` at the command line, it tells the MSBuild system to build the project and create a ClickOnce application in the publish folder. This is equivalent to selecting the **Publish** command in the IDE.

This command executes *msbuild.exe*, which is on the path in the Visual Studio command-prompt environment.

A "target" is an indicator to MSBuild on how to process the command. The key targets are the "build" target and the "publish" target. The build target is the equivalent to selecting the Build command (or pressing F5) in the IDE. If you only want to build your project, you can achieve that by typing `msbuild`. This command works because the build target is the default target for all projects generated by Visual Studio. This means you do not explicitly need to specify the build target. Therefore, typing `msbuild` is the same operation as typing

```
msbuild /target:build .
```

The `/target:publish` command tells MSBuild to invoke the publish target. The publish target depends on the build target. This means that the publish operation is a superset of the build operation. For example, if you made a change to one of your Visual Basic or C# source files, the corresponding assembly would automatically be rebuilt by the publish operation.

For information on generating a full ClickOnce deployment using the Mage.exe command-line tool to create your ClickOnce manifest, see [Walkthrough: Manually deploy a ClickOnce application](#).

## Create and build a basic ClickOnce application with MSBuild

### To create and publish a ClickOnce project

1. Open Visual Studio and create a new project.

Choose the **Windows Desktop Application** project template and name the project `CmdLineDemo`.

2. From the **Build** menu, click the **Publish** command.

This step ensures that the project is properly configured to produce a ClickOnce application deployment.

The Publish Wizard appears.

3. In the Publish Wizard, click **Finish**.

Visual Studio generates and displays the default Web page, called *Publish.htm*.

4. Save your project, and make note of the folder location in which it is stored.

The steps above create a ClickOnce project which has been published for the first time. Now you can

reproduce the build outside of the IDE.

**To reproduce the build from the command line**

1. Exit Visual Studio.
2. From the Windows **Start** menu, click **All Programs**, then **Microsoft Visual Studio**, then **Visual Studio Tools**, then **Visual Studio Command Prompt**. This should open a command prompt in the root folder of the current user.
3. In the **Visual Studio Command Prompt**, change the current directory to the location of the project you just built above. For example, type `chdir My Documents\Visual Studio\Projects\CmdLineDemo`.
4. To remove the existing files produced in "To create and publish a ClickOnce project," type `rmdir /s publish`.

This step is optional, but it ensures that the new files were all produced by the command-line build.

5. Type `msbuild /target:publish`.

The above steps will produce a full ClickOnce application deployment in a subfolder of your project named **Publish**. *CmdLineDemo.application* is the ClickOnce deployment manifest. The folder *CmdLineDemo\_1.0.0.0* contains the files *CmdLineDemo.exe* and *CmdLineDemo.exe.manifest*, the ClickOnce application manifest. *Setup.exe* is the bootstrapper, which by default is configured to install the .NET Framework. The DotNetFX folder contains the redistributables for the .NET Framework. This is the entire set of files you need to deploy your application over the Web or via UNC or CD/DVD.

**NOTE**

The MSBuild system uses the **PublishDir** option to specify the location for output, for example

```
msbuild /t:publish /p:PublishDir="<specific location>"
```

## Build .NET ClickOnce applications from the command line

Building .NET ClickOnce applications from the command line is a similar experience except, you need to provide an additional property for the publish profile on the MSBuild command line. The easiest way to create a publish profile is by using Visual Studio. See [Deploy a .NET Windows application using ClickOnce](#) for more information.

Once you have the publish profile created, you can provide the pubxml file as a property on the msbuild command line. For example:

```
msbuild /t:publish /p:PublishProfile=<pubxml file> /p:PublishDir="<specific location>"
```

## Publish properties

When you publish the application in the above procedures, the following properties are inserted into your project file by the Publish Wizard or in the publish profile file for .NET Core 3.1, or later projects. These properties directly influence how the ClickOnce application is produced.

In *CmdLineDemo.vbproj* / *CmdLineDemo.csproj*:

```
<AssemblyOriginatorKeyFile>WindowsApplication3.snk</AssemblyOriginatorKeyFile>
<GenerateManifests>true</GenerateManifests>
<TargetZone>LocalIntranet</TargetZone>
<PublisherName>Microsoft</PublisherName>
<ProductName>CmdLineDemo</ProductName>
<PublishUrl>http://localhost/CmdLineDemo</PublishUrl>
<Install>true</Install>
<ApplicationVersion>1.0.0.*</ApplicationVersion>
<ApplicationRevision>1</ApplicationRevision>
<UpdateEnabled>true</UpdateEnabled>
<UpdateRequired>false</UpdateRequired>
<UpdateMode>Foreground</UpdateMode>
<UpdateInterval>7</UpdateInterval>
<UpdateIntervalUnits>Days</UpdateIntervalUnits>
<UpdateUrlEnabled>false</UpdateUrlEnabled>
<IsWebBootstrapper>true</IsWebBootstrapper>
<BootstrapperEnabled>true</BootstrapperEnabled>
```

For .NET Framework projects, you can override any of these properties at the command line without altering the project file itself. For example, the following will build the ClickOnce application deployment without the bootstrapper:

```
msbuild /target:publish /property:BootstrapperEnabled=false
```

For .NET Core 3.1, or later, projects these settings are provided in the pubxml file.

Publishing properties are controlled in Visual Studio from the **Publish**, **Security**, and **Signing** property pages of the **Project Designer**. Below is a description of the publishing properties, along with an indication of how each is set in the various property pages of the application designer:

#### NOTE

For .NET Windows desktop projects, these settings are now found in the Publish Wizard

- **AssemblyOriginatorKeyFile** determines the key file used to sign your ClickOnce application manifests. This same key may also be used to assign a strong name to your assemblies. This property is set on the **Signing** page of the **Project Designer**.

For .NET windows applications, this setting remains in the project file

The following properties are set on the **Security** page:

- **Enable ClickOnce Security Settings** determines whether ClickOnce manifests are generated. When a project is initially created, ClickOnce manifest generation is off by default. The wizard will automatically turn this flag on when you publish for the first time.
- **TargetZone** determines the level of trust to be emitted into your ClickOnce application manifest. Possible values are "Internet", "LocalIntranet", and "Custom". Internet and LocalIntranet will cause a default permission set to be emitted into your ClickOnce application manifest. LocalIntranet is the default, and it basically means full trust. Custom specifies that only the permissions explicitly specified in the base *app.manifest* file are to be emitted into the ClickOnce application manifest. The *app.manifest* file is a partial manifest file that contains just the trust information definitions. It is a hidden file, automatically added to your project when you configure permissions on the **Security** page.

-

## NOTE

For .NET Core 3.1, or later, Windows desktop projects, these Security settings are not supported.

The following properties are set on the **Publish** page:

- `PublishUrl` is the location where the application will be published to in the IDE. It is inserted into the ClickOnce application manifest if neither the `InstallUrl` or `UpdateUrl` property is specified.
- `ApplicationVersion` specifies the version of the ClickOnce application. This is a four-digit version number. If the last digit is a "\*", then the `ApplicationRevision` is substituted for the value inserted into the manifest at build time.
- `ApplicationRevision` specifies the revision. This is an integer which increments each time you publish in the IDE. Notice that it is not automatically incremented for builds performed at the command-line.
- `Install` determines whether the application is an installed application or a run-from-Web application.
- `InstallUrl` (not shown) is the location where users will install the application from. If specified, this value is burned into the *setup.exe* bootstrapper if the `IsWebBootstrapper` property is enabled. It is also inserted into the application manifest if the `UpdateUrl` is not specified.
- `SupportUrl` (not shown) is the location linked in the **Add/Remove Programs** dialog box for an installed application.

The following properties are set in the **Application Updates** dialog box, accessed from the **Publish** page.

- `UpdateEnabled` indicates whether the application should check for updates.
- `UpdateMode` specifies either Foreground updates or Background updates.

For .NET Core 3.1, or later, projects, Background is not supported.

- `UpdateInterval` specifies how frequently the application should check for updates.

For .NET Core 3.1, or later, this setting is not supported.

- `UpdateIntervalUnits` specifies whether the `UpdateInterval` value is in units of hours, days, or weeks.

For .NET Core 3.1, or later, this setting is not supported.

- `UpdateUrl` (not shown) is the location from which the application will receive updates. If specified, this value is inserted into the application manifest.

The following properties are set in the **Publish Options** dialog box, accessed from the **Publish** page.

- `PublisherName` specifies the name of the publisher displayed in the prompt shown when installing or running the application. In the case of an installed application, it is also used to specify the folder name on the **Start** menu.
- `ProductName` specifies the name of the product displayed in the prompt shown when installing or running the application. In the case of an installed application, it is also used to specify the shortcut name on the **Start** menu.

The following properties are set in the **Prerequisites** dialog box, accessed from the **Publish** page.

- `BootstrapperEnabled` determines whether to generate the *setup.exe* bootstrapper.

- `IsWebBootstrapper` determines whether the *setup.exe* bootstrapper works over the Web or in disk-based mode.

## InstallURL, SupportUrl, PublishURL, and UpdateURL

The following table shows the four URL options for ClickOnce deployment.

| URL OPTION              | DESCRIPTION  |
|-------------------------|--|
| <code>PublishURL</code> | Required if you are publishing your ClickOnce application to a Web site.   |
| <code>InstallURL</code> | Optional. Set this URL option if the installation site is different than the <code>PublishURL</code> . For example, you could set the <code>PublishURL</code> to an FTP path and set the <code>InstallURL</code> to a Web URL. |
| <code>SupportURL</code> | Optional. Set this URL option if the support site is different than the <code>PublishURL</code> . For example, you could set the <code>SupportURL</code> to your company's customer support Web site.                          |
| <code>UpdateURL</code>  | Optional. Set this URL option if the update location is different than the <code>InstallURL</code> . For example, you could set the <code>PublishURL</code> to an FTP path and set the <code>UpdateURL</code> to a Web URL.    |

## See also

- [GenerateBootstrapper](#)
- [GenerateApplicationManifest](#)
- [GenerateDeploymentManifest](#)
- [ClickOnce security and deployment](#)
- [Walkthrough: Manually deploy a ClickOnce application](#)

# Publish an application to IIS by importing publish settings in Visual Studio

3/5/2021 • 7 minutes to read • [Edit Online](#)

You can use the **Publish** tool to import publish settings and then deploy your app. In this article, we use publish settings for IIS, but you can use similar steps to import publish settings for [Azure App Service](#). In some scenarios, use of a publish settings profile can be faster than manually configuring deployment to IIS for each installation of Visual Studio.

These steps apply to ASP.NET, ASP.NET Core, and .NET Core apps in Visual Studio.

In this tutorial, you will:

- Configure IIS so that you can generate a publish settings file
- Create a publish settings file
- Import the publish settings file into Visual Studio
- Deploy the app to IIS

A publish settings file (*\*.publishsettings*) is different than a publishing profile (*\*.pubxml*) created in Visual Studio. A publish settings file is created by IIS or Azure App Service, or it can be manually created, and then it can be imported into Visual Studio.

## NOTE

If you just need to copy a Visual Studio publishing profile (\*.pubxml file) from one installation of Visual Studio to another, you can find the publishing profile, *<profilename>.pubxml*, in the *\\<projectname>\\Properties\\PublishProfiles* folder for managed project types. For websites, look under the *\\App\_Data* folder. The publishing profiles are MSBuild XML files.

## Prerequisites

- You must have Visual Studio 2019 installed and the **ASP.NET and web development** workload.

If you haven't already installed Visual Studio, go to the [Visual Studio downloads](#) page to install it for free.

- You must have Visual Studio 2017 installed and the **ASP.NET and web development** workload.

If you haven't already installed Visual Studio, go to the [Visual Studio downloads](#) page to install it for free.

- On your server, you must be running Windows Server 2012, Windows Server 2016, or Windows Server 2019, and you must have the [IIS Web Server role](#) correctly installed (required to generate the publish settings file (*\*.publishsettings*)). Either ASP.NET 4.5 or ASP.NET Core must also be installed on the server. To set up ASP.NET 4.5, see [IIS 8.0 Using ASP.NET 3.5 and ASP.NET 4.5](#). To set up ASP.NET Core, see [Host ASP.NET Core on Windows with IIS](#). For ASP.NET Core, make sure you configure the Application Pool to use **No Managed Code**, as described in the article.

## Create a new ASP.NET project in Visual Studio

1. On the computer running Visual Studio, create a new project.

Choose the correct template. In this example, choose either **ASP.NET Web Application (.NET Framework)** or (for C# only) **ASP.NET Core Web Application**, and then select OK.



If you don't see the specified project templates, go to the **Open Visual Studio Installer** link in the left pane of the **New Project** dialog box. The Visual Studio Installer launches. Install the **ASP.NET and web development** workload.

The project template you choose (ASP.NET or ASP.NET Core) must correspond to the version of ASP.NET installed on the web server.

2. Choose either **MVC** (.NET Framework) or **Web Application (Model-View-Controller)** (for .NET Core), and make sure that **No Authentication** is selected, and then select **OK**.
3. Type a name like **MyWebApp** and select **OK**.

Visual Studio creates the project.

4. Choose **Build > Build Solution** to build the project.

## Install and configure Web Deploy on Windows Server

Web Deploy 3.6 for Hosting Servers provides additional configuration features that enable the creation of the publish settings file from the UI.

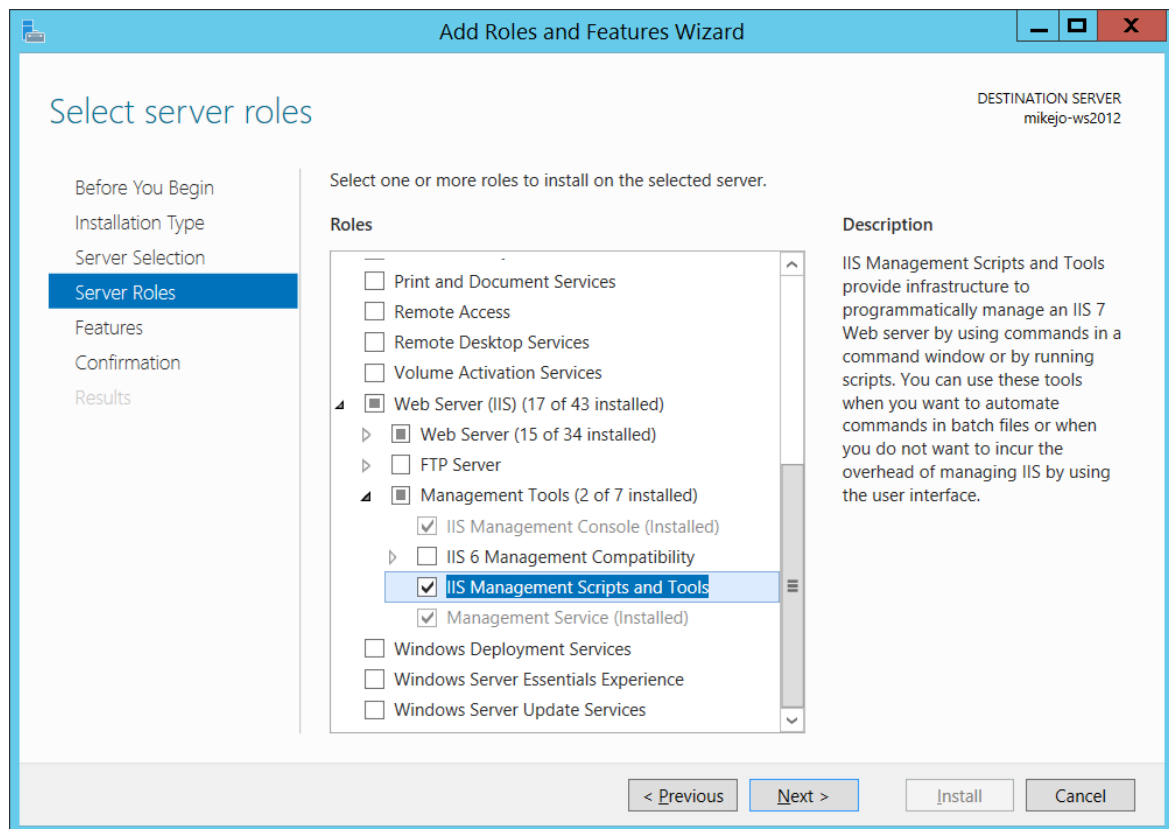
1. If you have Web Deploy already installed on Windows Server, uninstall it using **Control Panel > Programs > Uninstall a Program**.
2. Next, install Web Deploy 3.6 for Hosting Servers on Windows Server.

To install Web Deploy for Hosting Servers, use the Web Platform Installer (WebPI). (To find the Web Platform Installer link from IIS, select **IIS** in the left pane of Server Manager. In the server pane, right-click the server and select **Internet Information Services (IIS) Manager**. Then use the **Get New Web Platform Components** link in the **Actions** window.) You can also obtain the Web Platform Installer (WebPI) from [downloads](#).

In the Web Platform Installer, you find **Web Deploy 3.6 for Hosting Servers** in the Applications tab.

3. If you did not already install **IIS Management Scripts and Tools**, install it now.

Go to **Select server roles > Web Server (IIS) > Management Tools**, and then select the **IIS Management Scripts and Tools** role, click **Next**, and then install the role.



The scripts and tools are required to enable the generation of the publish settings file.

4. (Optional) Verify that Web Deploy is running correctly by opening **Control Panel > System and Security > Administrative Tools > Services**, and then make sure that:

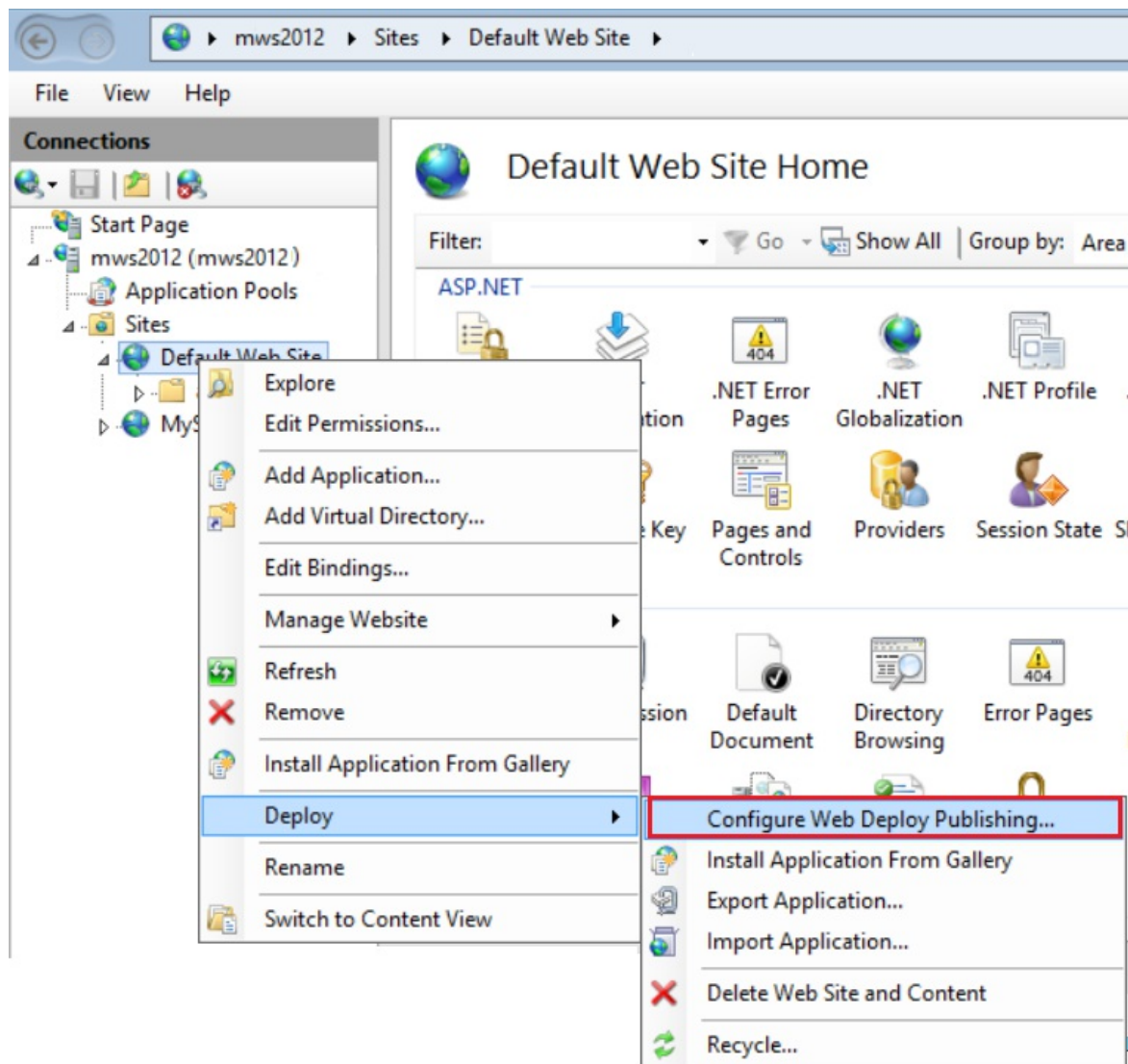
- **Web Deployment Agent Service** is running (the service name is different in older versions).
- **Web Management Service** is running.

If one of the agent services is not running, restart the **Web Deployment Agent Service**.

If the Web Deployment Agent Service is not present at all, go to **Control Panel > Programs > Uninstall a program**, find **Microsoft Web Deploy <version>**. Choose to **Change** the installation and make sure that you choose **Will be installed to the local hard drive** for the Web Deploy components. Complete the change installation steps.

## Create the publish settings file in IIS on Windows Server

1. Close and reopen the IIS Management Console to show updated configuration options in the UI.
2. In IIS, right-click the **Default Web Site**, choose **Deploy > Configure Web Deploy Publishing**.



If you don't see the **Deploy** menu, see the preceding section to verify that Web Deploy is running.

3. In the **Configure Web Deploy Publishing** dialog box, examine the settings.
4. Click **Setup**.

In the **Results** panel, the output shows that access rights are granted to the specified user, and that a file with a *.publishsettings* file extension has been generated in the location shown in the dialog box.

```
<?xml version="1.0" encoding="utf-8"?>
<publishData>
  <publishProfile
    publishUrl="https://myhostname:8172/msdeploy.axd"
    msdeploySite="Default Web Site"
    destinationAppUrl="http://myhostname:80/"
    mySQLDBConnectionString=""
    SQLServerDBConnectionString=""
    profileName="Default Settings"
    publishMethod="MSDeploy"
    userName="myhostname\myusername" />
</publishData>
```

Depending on your Windows Server and IIS configuration, you see different values in the XML file. Here are a few details about the values that you see:

- The *msdeploy.axd* file referenced in the `publishUrl` attribute is a dynamically generated HTTP handler file for Web Deploy. (For testing purposes, `http://myhostname:8172` generally works as

well.)

- The `publishUrl` port is set to port 8172, which is the default for Web Deploy.
- The `destinationAppUrl` port is set to port 80, which is the default for IIS.
- If you are unable to connect to the remote host in Visual Studio using the host name (in later steps), test the IP address in place of the host name.

#### NOTE

If you are publishing to IIS running on an Azure VM, you must open the Web Deploy and IIS ports in the Network Security group. For detailed information, see [Install and run IIS](#).

5. Copy this file to the computer where you are running Visual Studio.

## Import the publish settings in Visual Studio and deploy

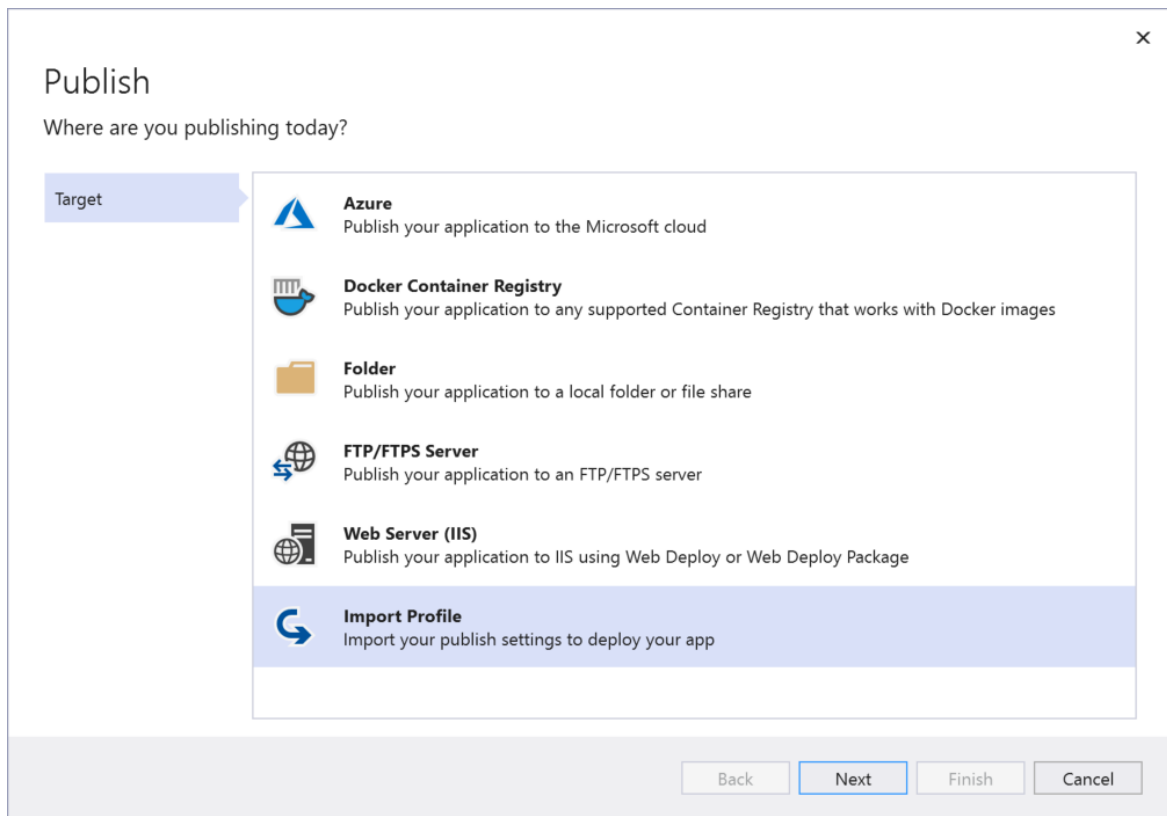
1. On the computer where you have the ASP.NET project open in Visual Studio, right-click the project in Solution Explorer, and choose **Publish**.

If you have previously configured any publishing profiles, the **Publish** pane appears. Click **New** or **Create new profile**.

2. Select the option to import a profile.

In the **Publish** dialog box, click **Import Profile**.

In the **Pick a publish target** dialog box, click **Import Profile**.



3. Navigate to the location of the publish settings file that you created in the previous section.
4. In the **Import Publish Settings File** dialog, navigate to and select the profile that you created in the previous section, and click **Open**.

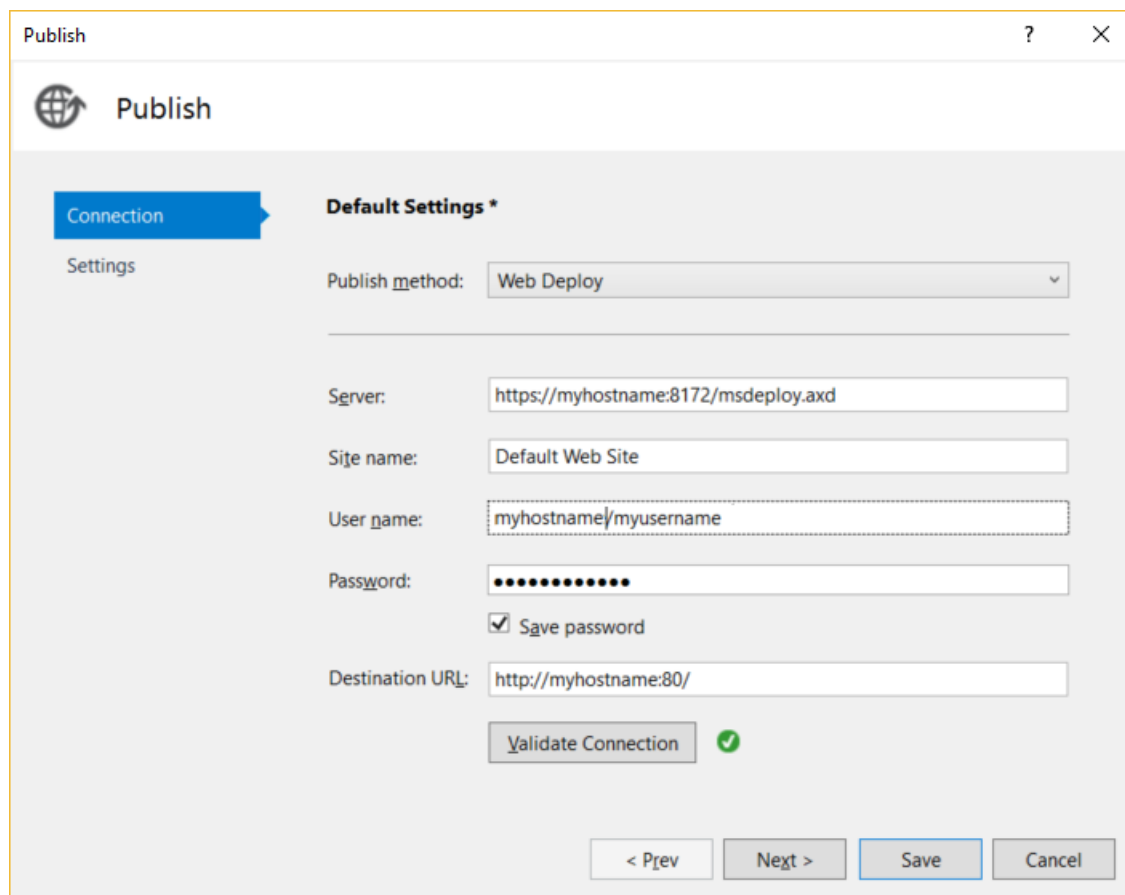
Click **Finish** to save the publishing profile, and then click **Publish**.

Visual Studio begins the deployment process, and the Output window shows progress and results.

If you get an any deployment errors, click **Edit** to edit settings. Modify settings and click **Validate** to test new settings. If the host name is not found, try the IP address instead of the host name in the **Server** and **Destination URL** fields.

Visual Studio begins the deployment process, and the Output window shows progress and results.

If you get an any deployment errors, click **Settings** to edit settings. Modify settings and click **Validate** to test new settings. If the host name is not found, try the IP address instead of the host name in the **Server** and **Destination URL** fields.



The screenshot shows the 'Publish' dialog box in Visual Studio, specifically the 'Default Settings \*' tab. The dialog has a title bar with 'Publish', a help icon, and a close icon. On the left, there's a sidebar with 'Connection' (selected) and 'Settings'. The main area contains the following fields and controls:

- Publish method:** A dropdown menu set to 'Web Deploy'.
- Server:** A text box containing 'https://myhostname:8172/msdeploy.axd'.
- Site name:** A text box containing 'Default Web Site'.
- User name:** A text box containing 'myhostname/myusername'.
- Password:** A text box filled with dots, with a checked checkbox labeled 'Save password' below it.
- Destination URL:** A text box containing 'http://myhostname:80/'.
- Validate Connection:** A button with a green checkmark icon to its right.
- Navigation buttons:** '< Prev', 'Next >', 'Save', and 'Cancel' at the bottom.

After the app deploys successfully, it should start automatically. If it does not start from Visual Studio, start the app in IIS. For ASP.NET Core, you need to make sure that the Application pool field for the **DefaultAppPool** is set to **No Managed Code**.

## Next steps

In this tutorial, you created a publish settings file, imported it into Visual Studio, and deployed an ASP.NET app to IIS. You may want an overview of other publishing options in Visual Studio.

[First look at deployment](#)

# Publish an application to Azure App Service by importing publish settings in Visual Studio

3/5/2021 • 4 minutes to read • [Edit Online](#)

You can use the **Publish** tool to import publish settings and then deploy your app. In this article, we use publish settings for Azure App Service, but you can use similar steps to import publish settings from [IIS](#). In some scenarios, use of a publish settings profile can be faster than manually configuring deployment to the service for each installation of Visual Studio.

These steps apply to ASP.NET, ASP.NET Core, and .NET Core apps in Visual Studio. You can also import publish settings for [Python](#) apps.

In this tutorial, you will:

- Generate a publish settings file from Azure App Service
- Import the publish settings file into Visual Studio
- Deploy the app to Azure App Service

A publish settings file (*\*.publishsettings*) is different than a publishing profile (*\*.pubxml*) created in Visual Studio. A publish settings file is created by Azure App Service, and then it can be imported into Visual Studio.

## NOTE

If you just need to copy a Visual Studio publishing profile (*\*.pubxml* file) from one installation of Visual Studio to another, you can find the publishing profile, *<profilename>.pubxml*, in the *\<projectname>\Properties\PublishProfiles* folder for managed project types. For websites, look under the *\App\_Data* folder. The publishing profiles are MSBuild XML files.

## Prerequisites

- You must have Visual Studio 2019 installed and the **ASP.NET and web development** workload.

If you haven't already installed Visual Studio, go to the [Visual Studio downloads](#) page to install it for free.

- You must have Visual Studio 2017 installed and the **ASP.NET and web development** workload.

If you haven't already installed Visual Studio, go to the [Visual Studio downloads](#) page to install it for free.

- Create an Azure App Service. For detailed instructions, see [Deploy an ASP.NET Core web app to Azure using Visual Studio](#).

## Create a new ASP.NET project in Visual Studio

1. On the computer running Visual Studio, create a new project.

Choose the correct template. In this example, choose either **ASP.NET Web Application (.NET Framework)** or (for C# only) **ASP.NET Core Web Application**, and then select **OK**.

If you don't see the specified project templates, go to the **Open Visual Studio Installer** link in the left pane of the **New Project** dialog box. The Visual Studio Installer launches. Install the **ASP.NET and web development** workload.

The project template you choose (ASP.NET or ASP.NET Core) must correspond to the version of ASP.NET

installed on the web server.

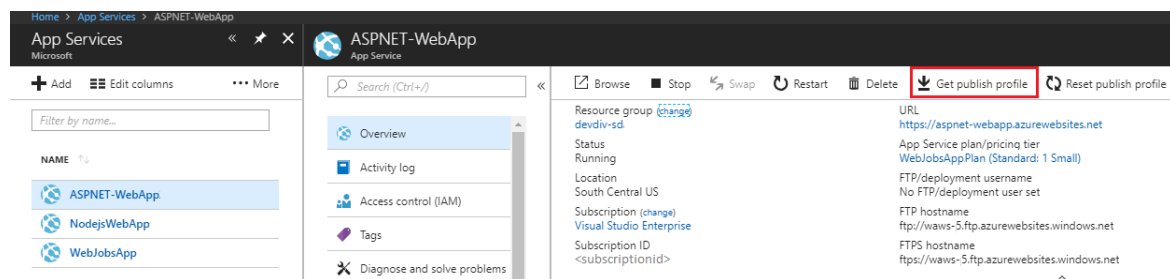
2. Choose either **MVC** (.NET Framework) or **Web Application (Model-View-Controller)** (for .NET Core), and make sure that **No Authentication** is selected, and then select **OK**.
3. Type a name like **MyWebApp** and select **OK**.

Visual Studio creates the project.

4. Choose **Build > Build Solution** to build the project.

## Create the publish settings file in Azure App Service

1. In the Azure portal, open the Azure App Service.
2. Go to **Get publish profile** and save the profile locally.



A file with a *.publishsettings* file extension has been generated in the location where you saved it. The following code shows a partial example of the file (in a more readable formatting).

```
<publishData>
  <publishProfile
    profileName="DeployASPDotNetCore - Web Deploy"
    publishMethod="MSDeploy"
    publishUrl="deployaspdotnetcore.scm.azurewebsites.net:443"
    msdeploySite="DeployASPDotNetCore"
    userName="$DeployASPDotNetCore"
    userPWD="abcdefghijklmnopqrstuvwxyz"
    destinationAppUrl="http://deployaspdotnetcore20180508031824.azurewebsites.net"
    SQLServerDBConnectionString=""
    mySQLDBConnectionString=""
    hostingProviderForumLink=""
    controlPanelLink="http://windows.azure.com"
    webSystem="WebSites">
    <databases />
  </publishProfile>
</publishData>
```

Typically, the preceding *\*.publishsettings* file contains two publishing profiles that you can use in Visual Studio, one to deploy using Web Deploy, and one to deploy using FTP. The preceding code shows the Web Deploy profile. Both profiles will be imported later when you import the profile.

## Import the publish settings in Visual Studio and deploy

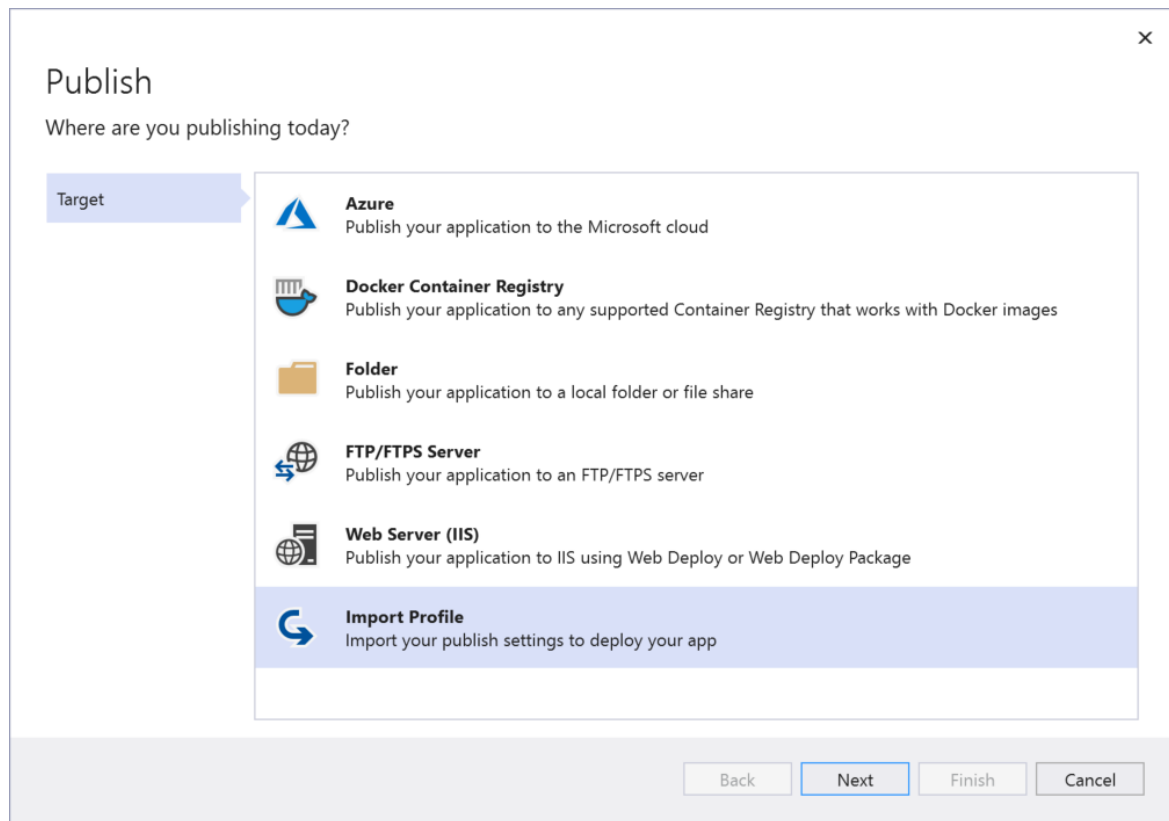
1. On the computer where you have the ASPNET project open in Visual Studio, right-click the project in Solution Explorer, and choose **Publish**.

If you have previously configured any publishing profiles, the **Publish** pane appears. Click **New** or **Create new profile**.

2. Select the option to import a profile.

In the **Publish** dialog box, click **Import Profile**.

In the **Pick a publish target** dialog box, click **Import Profile**.



3. Navigate to the location of the publish settings file that you created in the previous section.
4. In the **Import Publish Settings File** dialog, navigate to and select the profile that you created in the previous section, and click **Open**.

Click **Finish** to save the publishing profile, and then click **Publish**.

Visual Studio begins the deployment process, and the Output window shows progress and results.

If you get an any deployment errors, click **Edit** to edit settings. Modify settings and click **Validate** to test new settings. If the host name is not found, try the IP address instead of the host name in the **Server** and **Destination URL** fields.

Visual Studio begins the deployment process, and the Output window shows progress and results.


If you get an any deployment errors, click **Settings** to edit settings. Modify settings and click **Validate** to test new settings. If the host name is not found, try the IP address instead of the host name in the **Server** and **Destination URL** fields.



Publish

?

×

 Publish

Connection

Settings

Default Settings \*

Publish method:

Web Deploy

Server:

https://myhostname:8172/msdeploy.axd

Site name:

Default Web Site

User name:

myhostname/myusername

Password:

.....

☒ Save password

Destination URL:

http://myhostname:80/

Validate Connection

✓

< Prev

Next >

Save

Cancel

## Next steps

In this tutorial, you created a publish settings file, imported it into Visual Studio, and deployed an ASP.NET app to Azure App Service. You may want an overview of publishing options in Visual Studio.

[First look at deployment](#)

# First look at deployment in Visual Studio

3/5/2021 • 5 minutes to read • [Edit Online](#)

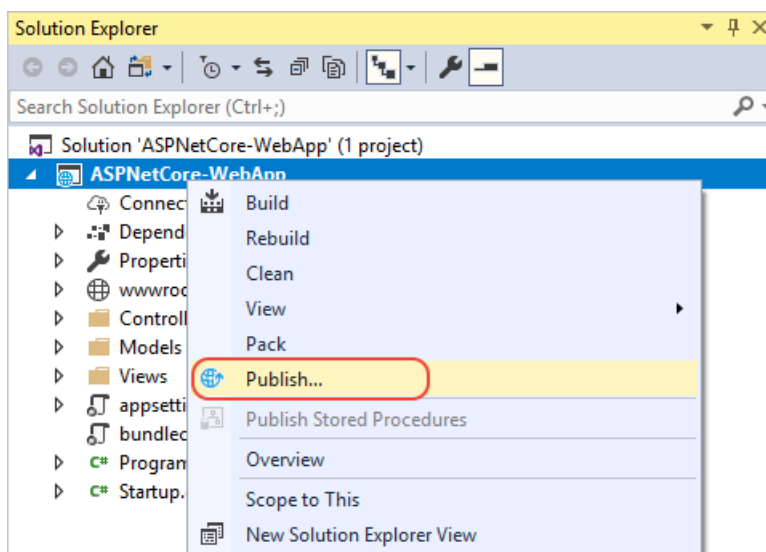
By deploying an application, service, or component, you distribute it for installation on other computers, devices, or servers, or in the cloud. You choose the appropriate method in Visual Studio for the type of deployment that you need. (Many app types support other deployment tools, such as command-line deployment or NuGet, that aren't described here.)

See the quickstarts and tutorials for step-by-step deployment instructions. For an overview of deployment options, see [What publishing options are right for me?](#).

## Deploy to a local folder

Deployment to a local folder is typically used for testing or to begin a staged deployment in which another tool is used for final deployment.

- **ASP.NET, ASP.NET Core, Node.js, Python, and .NET Core:** Use the **Publish** tool to deploy to a local folder. The exact options available depend on your app type. In Solution Explorer, right-click your project and select **Publish**. (If you haven't previously configured any publishing profiles, you must then select **Create new profile**.) Next, select **Folder**. For more information, see [Deploy to a local folder](#).



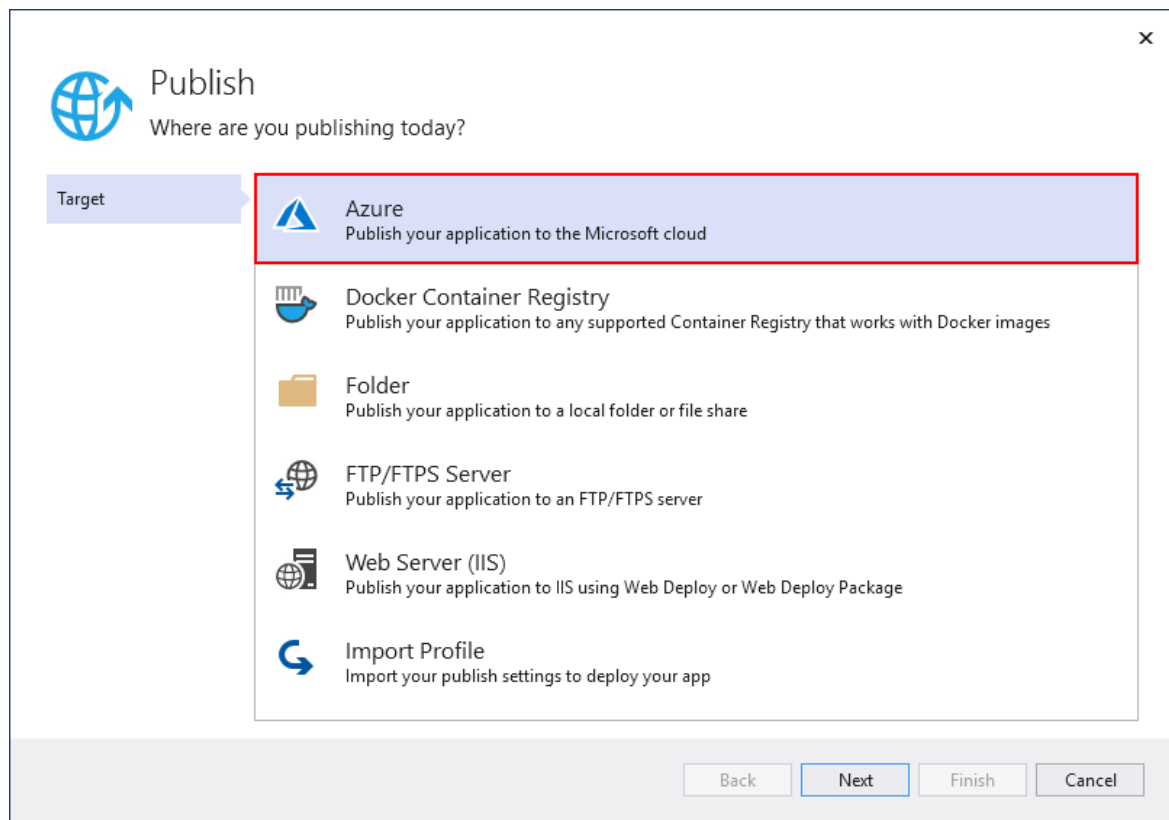
- **Windows desktop:** You can publish a Windows desktop application to a folder by using ClickOnce deployment. Users can then install the application with a single click. For more information, see the following articles:
  - [Deploy a .NET Framework Windows desktop app using ClickOnce](#).
  - [Deploy a .NET Windows desktop app using ClickOnce](#).
  - [Deploy a C++/CLR app using ClickOnce](#) or, for C/C++, see [Deploy a native app using a Setup project](#).

## Publish to Azure

- **ASP.NET, ASP.NET Core, Python, and Node.js:** Publish to Azure App Service or Azure App Service on Linux (using containers) by using one of the following methods:
  - For continuous (or automated) deployment of apps, use Azure DevOps with [Azure Pipelines](#).
  - For one-time (or manual) deployment of apps, use the **Publish** tool in Visual Studio.

For deployment that provides more customized configuration of the server, you can also use the **Publish** tool to deploy apps to an Azure virtual machine.

To use the **Publish** tool, right-click the project in Solution Explorer and select **Publish**. (If you've previously configured any publishing profiles, you must then select **Create new profile**.) In the **Publish** dialog box, select either **App Service** or **Azure Virtual Machines**, and then follow the configuration steps.



Starting in Visual Studio 2017 version 15.7, you can deploy ASP.NET Core apps to App Service on Linux.

For Python apps, also see [Python - Publishing to Azure App Service](#).

For a quick introduction, see [Publish to Azure](#) and [Publish to Linux](#). Also, see [Publish an ASP.NET Core app to Azure](#). For deployment using Git, see [Continuous deployment of ASP.NET Core to Azure with Git](#).

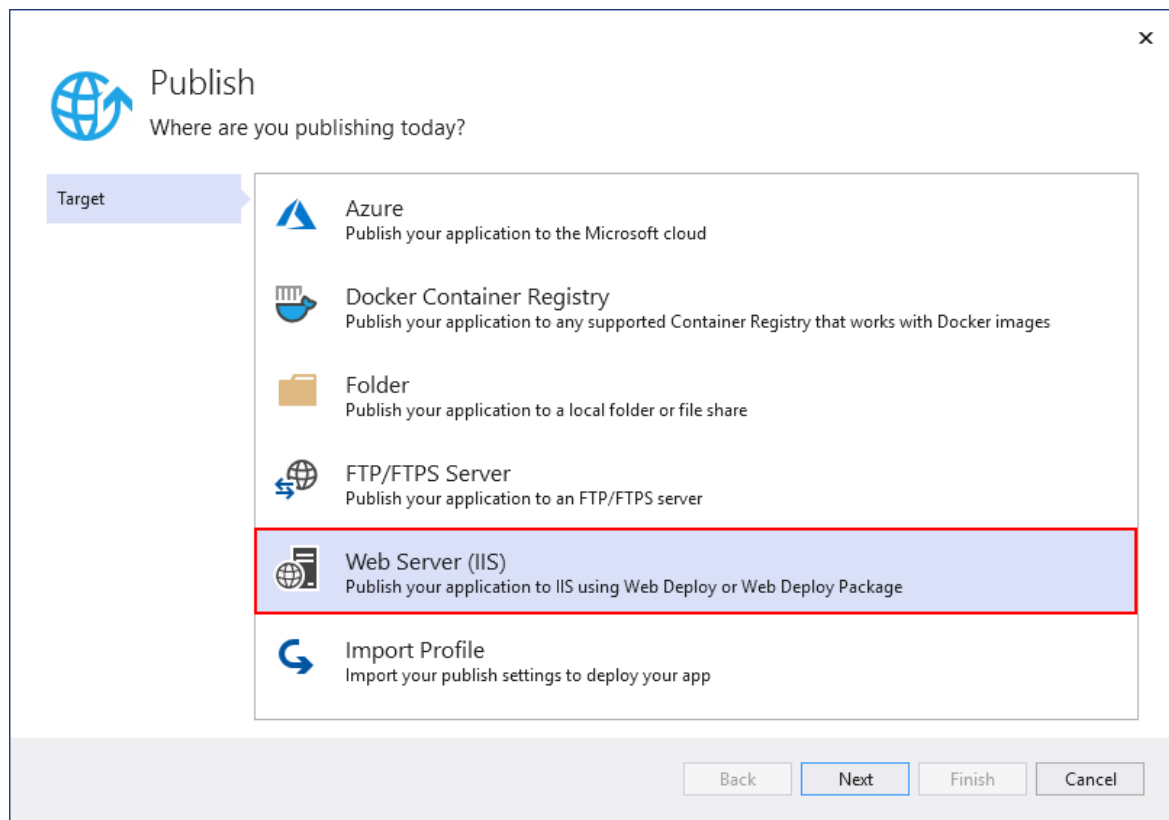
#### NOTE

If you don't already have an Azure account, you can [sign up here](#).

## Publish to the web or deploy to a network share

- **ASP.NET, ASP.NET Core, Node.js, and Python:** You can use the **Publish** tool to deploy to a website by using FTP or Web Deploy. For more information, see [Deploy to a website](#).

In Solution Explorer, right-click the project and select **Publish**. (If you've previously configured any publishing profiles, you must then select **Create new profile**.) In the **Publish** tool, select the option you want and follow the configuration steps.



For information on importing a publish profile in Visual Studio, see [Import publish settings and deploy to IIS](#).

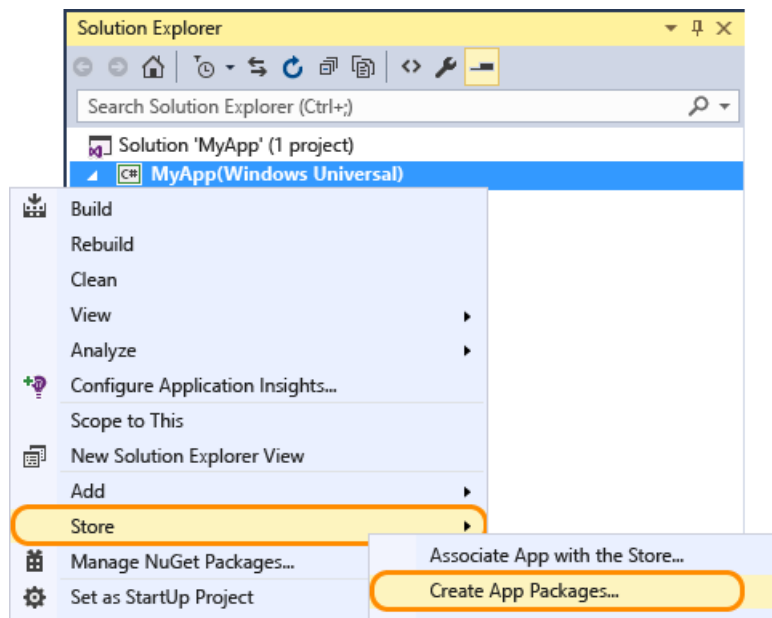
You can also deploy ASP.NET applications and services in a number of other ways. For more information, see [Deploying ASP.NET web applications and services](#).

- **Windows desktop:** You can publish a Windows desktop application to a web server or a network file share by using ClickOnce deployment. Users can then install the application with a single click. For more information, see the following articles:
  - [Deploy a .NET Framework Windows desktop app using ClickOnce](#)
  - [Deploy a .NET Windows desktop app using ClickOnce](#)
  - [Deploy a C++/CLR app using ClickOnce](#)

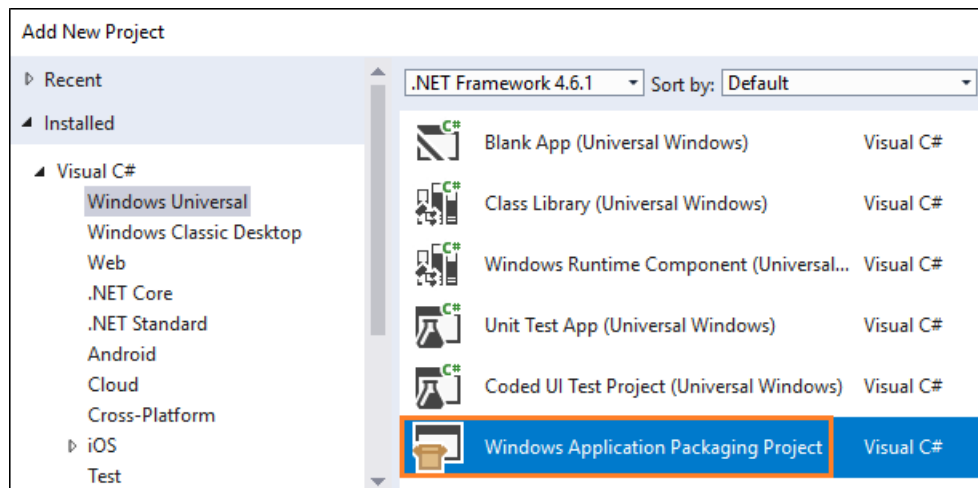
## Publish to Microsoft Store

From Visual Studio, you can create app packages for deployment to Microsoft Store.

- **UWP:** You can package your app and deploy it by using menu items. For more information, see [Package a UWP app by using Visual Studio](#).



- **Windows desktop:** You can deploy to Microsoft Store by using the Desktop Bridge starting in Visual Studio 2017 version 15.4. To do this, start by creating a Windows Application Packaging Project. For more information, see [Package a desktop app for Microsoft Store \(Desktop Bridge\)](#).



## Deploy to a device (UWP)

If you're deploying a UWP app for testing on a device, see [Run UWP apps on a remote machine in Visual Studio](#).

## Create an installer package (Windows desktop)

If you require a more complex installation of a desktop application than ClickOnce can provide, you can create a Windows Installer package (MSI or EXE installation file) or a custom bootstrapper.

- An MSI-based installer package can be created by using the [WiX Toolset Visual Studio 2017 Extension](#). This is a command-line toolset.
- An MSI or EXE installer package can be created by using [InstallShield](#) from Flexera Software. InstallShield may be used with Visual Studio 2017 and later versions. Community Edition isn't supported.

### NOTE

InstallShield Limited Edition is no longer included with Visual Studio and isn't supported in Visual Studio 2017 and later versions. Check with [Flexera Software](#) about future availability.

- An MSI or EXE installer package can be created by using a Setup project (vdproj). To use this option, install the [Visual Studio Installer Projects extension](#).
- You can also install prerequisite components for desktop applications by configuring a generic installer, which is known as a bootstrapper. For more information, see [Application deployment prerequisites](#).

## Deploy to a test lab

You can enable more sophisticated development and testing by deploying your applications into virtual environments. For more information, see [Test on a lab environment](#).

## Continuous deployment

You can use Azure Pipelines to enable continuous deployment of your app. For more information, see [Azure Pipelines](#) and [Deploy to Azure](#).

## Deploy a SQL database

- [Change target platform and publish a database project \(SQL Server Data Tools \(SSDT\)\)](#)
- [Deploy an Analysis Services Project \(SSAS\)](#)
- [Deploy Integration Services \(SSIS\) projects and packages](#)
- [Build and deploy to a local database](#)

## Deployment for other app types

| APP TYPE             | DEPLOYMENT SCENARIO   | LINK   |
|----------------------|---|--|
| Office app           | You can publish an add-in for Office from Visual Studio.  | <a href="#">Deploy and publish your Office add-in</a>      |
| WCF or OData service | Other applications can use WCF RIA services that you deploy to a web server.  | <a href="#">Developing and deploying WCF Data Services</a> |
| LightSwitch          | LightSwitch is no longer supported starting in Visual Studio 2017, but can still be deployed from Visual Studio 2015 and earlier. | <a href="#">Deploying LightSwitch applications</a>         |

## Next steps

In this tutorial, you took a quick look at deployment options for different applications.

[What publishing options are right for me?](#)

# Publish a Node.js application to Azure (Linux App Service)

3/5/2021 • 6 minutes to read • [Edit Online](#)

This tutorial walks you through the task of creating a simple Node.js application and publishing it to Azure.

When publishing a Node.js application to Azure, there are several options. These include Azure App Service, a VM running an OS of your choosing, Azure Container Service (AKS) for management with Kubernetes, a Container Instance using Docker, and more. For more details on each of these options, see [Compute](#).

For this tutorial, you deploy the app to [Linux App Service](#). Linux App Service deploys a Linux Docker container to run the Node.js application (as opposed to the Windows App Service, which runs Node.js apps behind IIS on Windows).

This tutorial shows how to create a Node.js application starting from a template installed with the Node.js Tools for Visual Studio, push the code to a repository on GitHub, and then provision an Azure App Service via the Azure web portal so that you can deploy from the GitHub repository. To use the command-line to provision the Azure App Service and push the code from a local Git repository, see [Create Node.js App](#).

In this tutorial, you learn how to:

- Create a Node.js project
- Create a GitHub repository for the code
- Create a Linux App Service on Azure
- Deploy to Linux

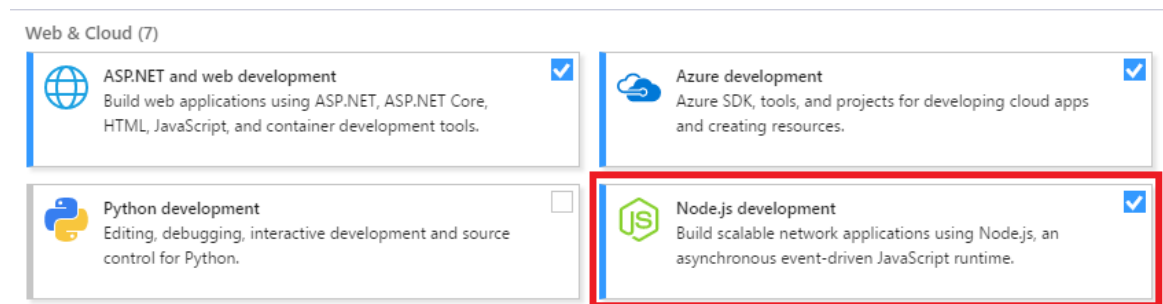
## Prerequisites

- You must have Visual Studio installed and the Node.js development workload.

If you haven't already installed Visual Studio 2019, go to the [Visual Studio downloads](#) page to install it for free.

If you haven't already installed Visual Studio 2017, go to the [Visual Studio downloads](#) page to install it for free.

If you need to install the workload but already have Visual Studio, go to **Tools > Get Tools and Features...**, which opens the Visual Studio Installer. Choose the **Node.js development** workload, then choose **Modify**.



- You must have the Node.js runtime installed.

If you don't have it installed, install the LTS version from the [Node.js](#) website. In general, Visual Studio

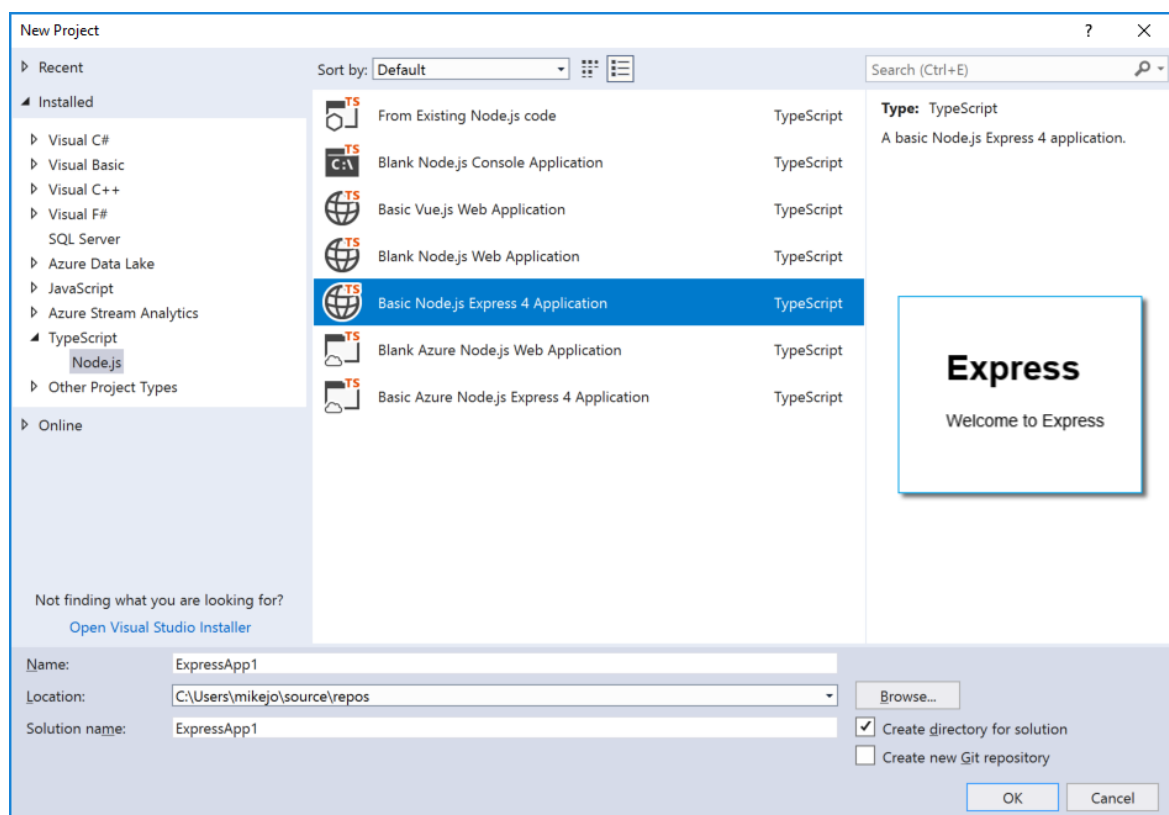
automatically detects the installed Node.js runtime. If it does not detect an installed runtime, you can configure your project to reference the installed runtime in the properties page (after you create a project, right-click the project node and choose **Properties**).

## Create a Node.js project to run in Azure

1. Open Visual Studio.
2. Create a new TypeScript Express app.

Press **Esc** to close the start window. Type **Ctrl + Q** to open the search box, type **Node.js**, then choose **Create new Basic Azure Node.js Express 4 application** (TypeScript). In the dialog box that appears, choose **Create**.

From the top menu bar, choose **File > New > Project**. In the left pane of the **New Project** dialog box, expand **TypeScript**, then choose **Node.js**. In the middle pane, choose **Basic Azure Node.js Express 4 application**, then choose **OK**.



If you don't see the **Basic Azure Node.js Express 4 application** project template, you must add the **Node.js development** workload. For detailed instructions, see the [Prerequisites](#).

Visual Studio creates the project and opens it in Solution Explorer (right pane).

3. Press **F5** to build and run the app, and make sure that everything is running as expected.
4. Select **File > Add to source control** to create a local Git repository for the project.

At this point, a Node.js app using the Express framework and written in TypeScript is working and checked in to local source control.

5. Edit the project as desired before proceeding to the next steps.

## Push code from Visual Studio to GitHub

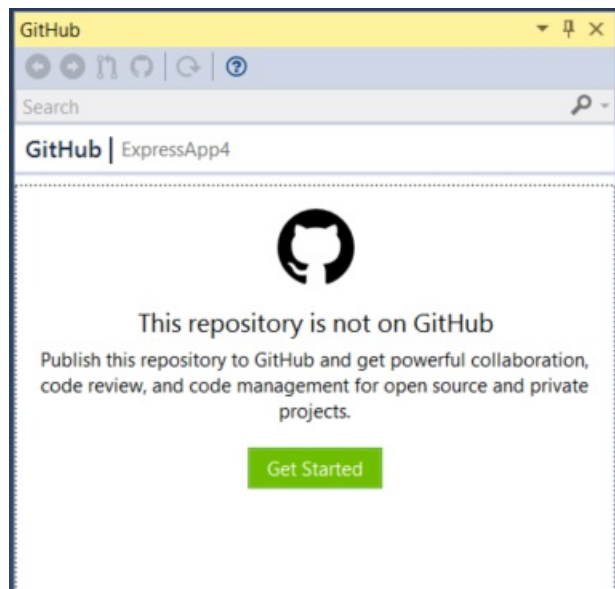
To set up GitHub for Visual Studio:



1. Make sure the [GitHub Extension for Visual Studio](#) is installed and enabled using the menu item **Tools > Extensions and Updates**.
2. From the menu select **View > Other Windows > GitHub**.

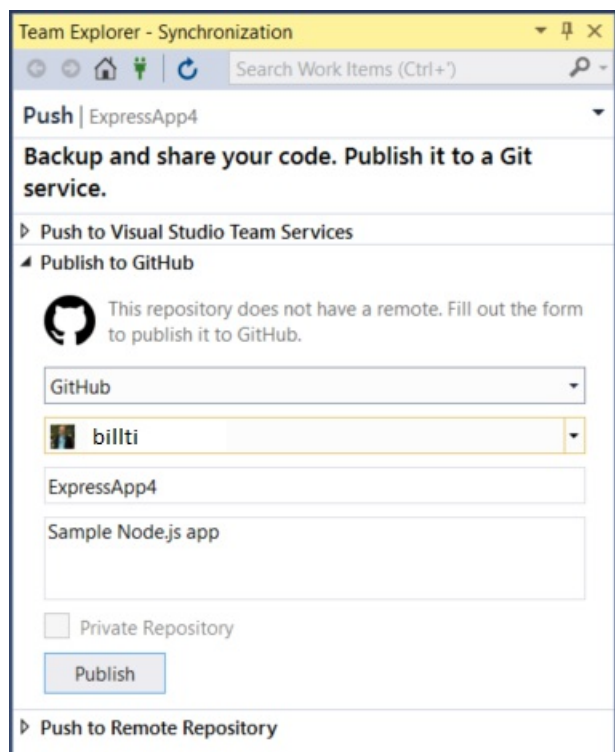
The GitHub window opens.

3. If you don't see the **Get Started** button in the GitHub window, click **File > Add to Source Control** and wait for the UI to update.



4. Click **Get started**.

If you are already connected to GitHub, the toolbox appears similar to the following illustration.



5. Complete the fields for the new repository to publish, and then click **Publish**.

After a few moments, a banner stating "Repository created successfully" appears.

In the next section, you learn how to publish from this repository to an Azure App Service on Linux.

# Create a Linux App Service in Azure

1. Sign in to the [Azure portal](#).
2. Select **App Services** from the list of services on the left, and then click **Add**.
3. If required, create a new Resource Group and App Service plan to host the new app.
4. Make sure to set the **OS** to **Linux**, and set **Runtime Stack** to the required Node.js version, as shown in the illustration.

Microsoft Azure

Home > App Services > Web + Mobile > Web App > Web App

Web App

Create

\* App name  
nodejs-linux ✓  
.azurewebsites.net

\* Subscription  
Visual Studio Ultimate with MSDN

\* Resource Group ⓘ  
☒ Create new ☐ Use existing  
nodejs-linux-rg ✓

\* OS  
☐ Windows ☒ Linux

\* App Service plan/Location  
nodejs-linux-plan(West US 2)

\* Runtime Stack  
Node.js 8.9

5. Click **Create** to create the App Service.  
It may take a few minutes to deploy.
6. After it is deployed, go to the **Application settings** section, and add a setting with a name of `SCM_SCRIPT_GENERATOR_ARGS` and a value of `--node`.

SETTINGS

Application settings

Authentication / Authorization

Managed service identity

Backups

Custom domains

SSL certificates

Networking

Runtime

Stack ⓘ Node.js 8.9

Startup File ⓘ

Application settings

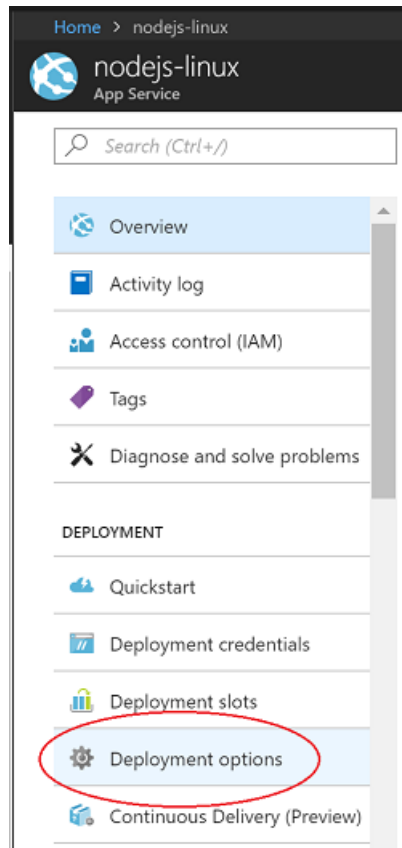
SCM\_SCRIPT\_GENERATOR\_ARGS --node

+ Add new setting

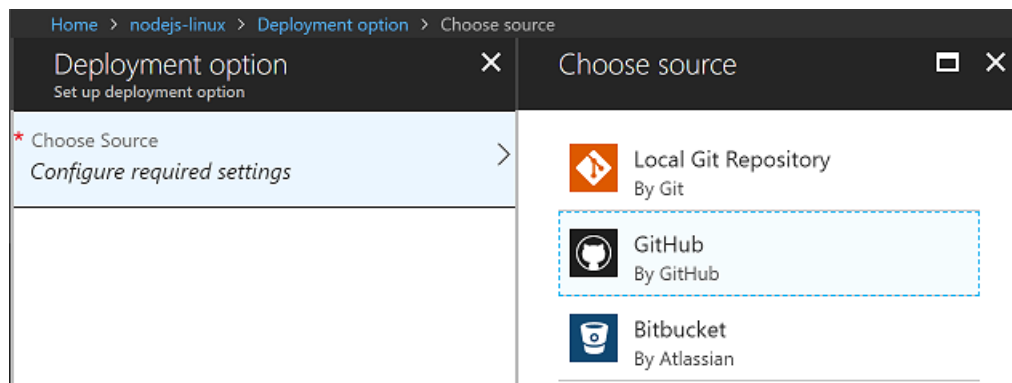
### WARNING

The App Service deployment process uses a set of heuristics to determine which type of application to try and run. If a `.sln` file is detected in the deployed content, it will assume an MSBuild based project is being deployed. The setting added above overrides this logic and specifies explicitly that this is a Node.js application. Without this setting, the Node.js application will fail to deploy if the `.sln` file is part of the repository being deployed to the App Service.

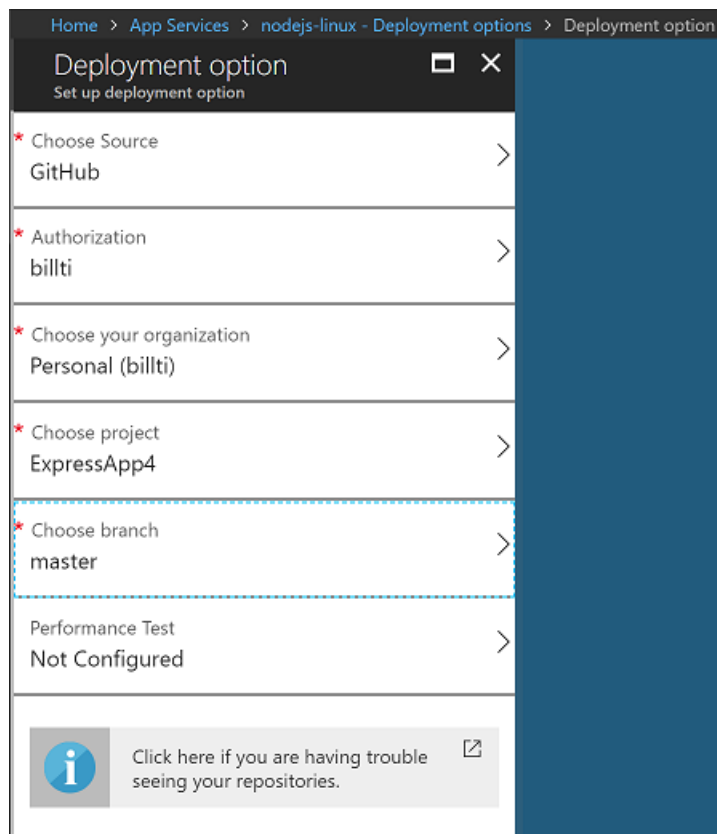
- Under **Application settings**, add another setting with a name of `WEBSITE_NODE_DEFAULT_VERSION` and a value of `8.9.0`.
- After it is deployed, open the App Service and select **Deployment options**.



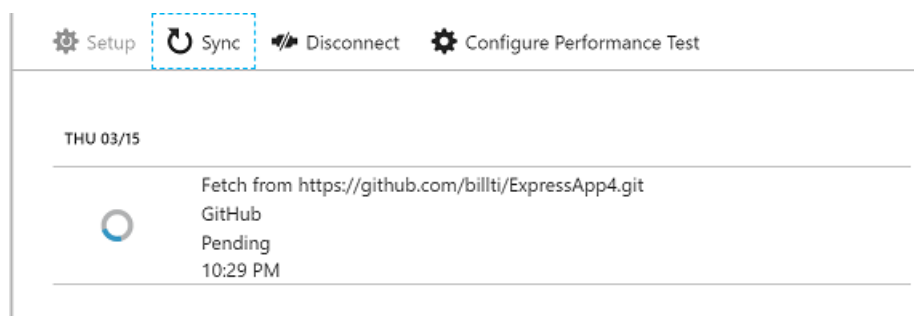
- Click **Choose source**, and then choose **GitHub**, and then configure any required permissions.



- Select the repository and branch to publish, and then select **OK**.



The **deployment options** page appears while syncing.



Once it is finished syncing, a check mark will appear.

The site is now running the Node.js application from the GitHub repository, and it is accessible at the URL created for the Azure App Service (by default the name given to the Azure App Service followed by ".azurewebsites.net").

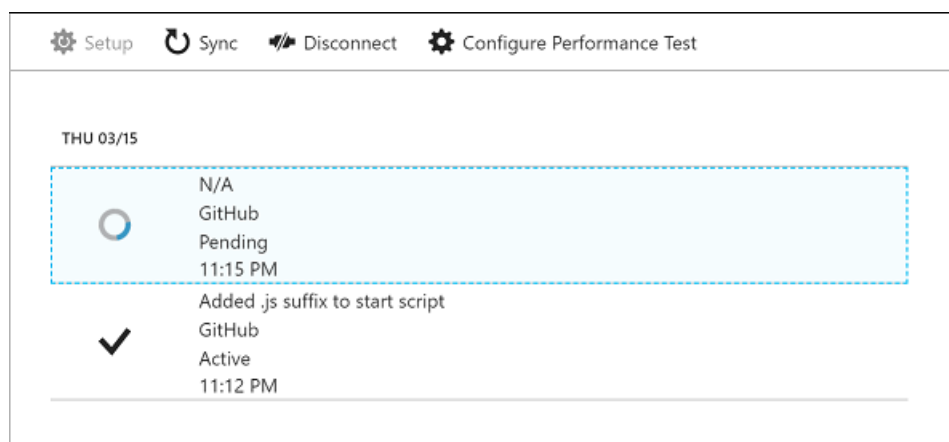
## Modify your app and push changes

1. Add the code shown here in *app.ts* after the line `app.use('/users', users);`. This adds a REST API at the URL */api*.

```
app.use('/api', (req, res, next) => {
  res.json({"result": "success"});
});
```

2. Build the code and test it locally, then check it in and push to GitHub.

In the Azure portal, it takes a few moments to detect changes in the GitHub repo, and then a new sync of the deployment starts. This looks similar to the following illustration.



- Once deployment is complete, navigate to the public site and append `/api` to the URL. The JSON response gets returned.

## Troubleshooting

- If the `node.exe` process dies (that is, an unhandled exception occurs), the container restarts.
- When the container starts up, it runs through various heuristics to figure out how to start the Node.js process. Details of the implementation can be seen at [generateStartupCommand.js](#).
- You can connect to the running container via SSH for investigations. This is easily done using the Azure portal. Select the App Service, and scroll down the list of tools until reaching SSH under the **Development Tools** section.
- To aid in troubleshooting, go to the **Diagnostics logs** settings for the App Service, and change the **Docker Container logging** setting from **Off** to **File System**. Logs are created in the container under `/home/LogFiles/_docker.log*`, and can be accessed on the box using SSH or FTP(S).
- A custom domain name may be assigned to the site, rather than the `*.azurewebsites.net` URL assigned by default. For more details, see the topic [Map Custom Domain](#).
- Deploying to a staging site for further testing before moving into production is a best practice. For details on how to configure this, see the topic [Create staging environments](#).
- See the [App Service on Linux FAQ](#) for more commonly asked questions.

## Next steps

In this tutorial, you learned how create a Linux App Service and deploy a Node.js application to the service. You may want to learn more about Linux App Service.

[Linux App Service](#)

# Publish to Azure App Service

3/5/2021 • 2 minutes to read • [Edit Online](#)

At present, Python is supported on Azure App Service for Linux, and you can publish apps using [Git deploy](#) and [containers](#), as described in this article.

## NOTE

Python support on Azure App Service for Windows is officially deprecated. As a result, the **Publish** command in Visual Studio is officially supported only for an [IIS target](#), and remote debugging on Azure App Service is no longer officially supported.

However, [Publishing to App Service on Windows](#) features still works for the time being, as the Python extensions for App Service on Windows remain available but will not be serviced or updated.

## Publish to App Service on Linux using Git deploy

Git deploy connects an App Service on Linux to a specific branch of a Git repository. Committing code to that branch automatically deploys to the App Service, and App Service automatically installs any dependencies listed in *requirements.txt*. In this case, App Service on Linux runs your code in a pre-configured container image that uses the Gunicorn web server. At present, this service is in Preview and not supported for production use.

For more information, see the following articles in the Azure documentation:

- [Quickstart: Create a Python web app in App Service](#) provides a short walkthrough of the Git deploy process using a simple Flask app and deployment from a local Git repository.
- [How to configure Python](#) describes the characteristics of the App Service on Linux container and how to customize the Gunicorn startup command for your app.

## Publish to App Service on Linux using containers

Instead of relying on the pre-built container with App Service on Linux, you can provide your own container. This option allows you to choose which web servers you use and to customize the container's behavior.

There are two options to build, manage, and push containers:

- Use Visual Studio Code and the Docker extension, as described on [Deploy Python using Docker Containers](#). Even if you don't use Visual Studio Code, the article provides helpful details on building container images for Flask and Django apps using the production-ready uwsgi and nginx web servers. You can then deploy those same container using the Azure CLI.
- Use the command line and Azure CLI, as described on [Use a custom Docker image](#) in the Azure documentation. This guide is generic, however, and not specific to Python.

## Publish to IIS

From Visual Studio, you can publish to a Windows virtual machine or other IIS-capable computer with the **Publish** command. When using IIS, be sure to create or modify a *web.config* file in the app that tells IIS where to find the Python interpreter. For more information, see [Configure web apps for IIS](#).

# ClickOnce security and deployment

3/5/2021 • 8 minutes to read • [Edit Online](#)

ClickOnce is a deployment technology that enables you to create self-updating Windows-based applications that can be installed and run with minimal user interaction. Visual Studio provides full support for publishing and updating applications deployed with ClickOnce technology if you have developed your projects with Visual Basic and Visual C#. For information about deploying Visual C++ applications, see [ClickOnce Deployment for Visual C++ Applications](#).

ClickOnce deployment overcomes three major issues in deployment:

- **Difficulties in updating applications.** With Microsoft Windows Installer deployment, whenever an application is updated, the user can install an update, an msp file, and apply it to the installed product; with ClickOnce deployment, you can provide updates automatically. Only those parts of the application that have changed are downloaded, and then the full, updated application is reinstalled from a new side-by-side folder.
- **Impact to the user's computer.** With Windows Installer deployment, applications often rely on shared components, with the potential for versioning conflicts; with ClickOnce deployment, each application is self-contained and cannot interfere with other applications.
- **Security permissions.** Windows Installer deployment requires administrative permissions and allows only limited user installation; ClickOnce deployment enables non-administrative users to install and grants only those Code Access Security permissions necessary for the application.

In the past, these issues sometimes caused developers to decide to create Web applications instead of Windows-based applications, sacrificing a rich user interface for ease of installation. By using applications deployed using ClickOnce, you can have the best of both technologies.

## What is a ClickOnce application?

A ClickOnce application is any Windows Presentation Foundation (.xbap), Windows Forms (.exe), console application (.exe), or Office solution (.dll) published using ClickOnce technology. You can publish a ClickOnce application in three different ways: from a Web page, from a network file share, or from media such as a CD-ROM. A ClickOnce application can be installed on an end user's computer and run locally even when the computer is offline, or it can be run in an online-only mode without permanently installing anything on the end user's computer. For more information, see [Choose a ClickOnce deployment strategy](#).

ClickOnce applications can be self-updating; they can check for newer versions as they become available and automatically replace any updated files. The developer can specify the update behavior; a network administrator can also control update strategies, for example, marking an update as mandatory. Updates can also be rolled back to an earlier version by the end user or by an administrator. For more information, see [Choose a ClickOnce update strategy](#).

Because ClickOnce applications are isolated, installing or running a ClickOnce application cannot break existing applications. ClickOnce applications are self-contained; each ClickOnce application is installed to and run from a secure per-user, per-application cache. ClickOnce applications run in the Internet or Intranet security zones. If necessary, the application can request elevated security permissions. For more information, see [Secure ClickOnce applications](#).

## How ClickOnce security works

The core ClickOnce security is based on certificates, code access security policies, and the ClickOnce trust prompt.

### **Certificates**

Authenticode certificates are used to verify the authenticity of the application's publisher. By using Authenticode for application deployment, ClickOnce helps prevent a harmful program from portraying itself as a legitimate program coming from an established, trustworthy source. Optionally, certificates can also be used to sign the application and deployment manifests to prove that the files have not been tampered with. For more information, see [ClickOnce and Authenticode](#). Certificates can also be used to configure client computers to have a list of trusted publishers. If an application comes from a trusted publisher, it can be installed without any user interaction. For more information, see [Trusted application deployment overview](#).

### **Code access security**

Code access security helps limit the access that code has to protected resources. In most cases, you can choose the Internet or Local Intranet zones to limit the permissions. Use the **Security** page in the **ProjectDesigner** to request the zone appropriate for the application. You can also debug applications with restricted permissions to emulate the end-user experience. For more information, see [Code access security for ClickOnce applications](#).

### **ClickOnce trust prompt**

If the application requests more permissions than the zone allows, the end user can be prompted to make a trust decision. The end user can decide if ClickOnce applications such as Windows Forms applications, Windows Presentation Foundation applications, console applications, XAML browser applications, and Office solutions are trusted to run. For more information, see [How to: Configure the ClickOnce trust prompt behavior](#).

## **How ClickOnce deployment works**

The core ClickOnce deployment architecture is based on two XML manifest files: an application manifest and a deployment manifest. The files are used to describe where the ClickOnce applications are installed from, how they are updated, and when they are updated.

### **Publish ClickOnce applications**

The application manifest describes the application itself. This includes the assemblies, the dependencies and files that make up the application, the required permissions, and the location where updates will be available. The application developer authors the application manifest by using the Publish Wizard in Visual Studio or the Manifest Generation and Editing Tool (*Mage.exe*) in the Windows Software Development Kit (SDK). For more information, see [How to: Publish a ClickOnce application using the Publish Wizard](#).

The deployment manifest describes how the application is deployed. This includes the location of the application manifest, and the version of the application that clients should run.

### **Deploy ClickOnce applications**

After it is created, the deployment manifest is copied to the deployment location. This can be a Web server, network file share, or media such as a CD. The application manifest and all the application files are also copied to a deployment location that is specified in the deployment manifest. This can be the same as the deployment location, or it can be a different location. When using the **Publish Wizard** in Visual Studio, the copy operations are performed automatically.

### **Install ClickOnce applications**

After it is deployed to the deployment location, end users can download and install the application by clicking an icon representing the deployment manifest file on a Web page or in a folder. In most cases, the end user is presented with a simple dialog box asking the user to confirm installation, after which installation proceeds and the application is started without additional intervention. In cases where the application requires elevated permissions or if the application is not signed by a trusted certificate, the dialog box also asks the user to grant permission before the installation can continue. Though ClickOnce installs are per-user, permission elevation



may be required if there are prerequisites that require administrator privileges. For more information about elevated permissions, see [Securing ClickOnce applications](#).

Certificates can be trusted at the machine or enterprise level, so that ClickOnce applications signed with a trusted certificate can install silently. For more information about trusted certificates, see [Trusted application deployment overview](#).

The application can be added to the user's **Start** menu and to the **Add or Remove Programs** group in the **Control Panel**. Unlike other deployment technologies, nothing is added to the **Program Files** folder or the registry, and no administrative rights are required for installation

#### NOTE

It is also possible to prevent the application from being added to the **Start** menu and **Add or Remove Programs** group, in effect making it behave like a Web application. For more information, see [Choose a ClickOnce deployment strategy](#).

### Update ClickOnce applications

When the application developers create an updated version of the application, they generate a new application manifest and copy files to a deployment location—usually a sibling folder to the original application deployment folder. The administrator updates the deployment manifest to point to the location of the new version of the application.

#### NOTE

The **Publish Wizard** in Visual Studio can be used to perform these steps.

In addition to the deployment location, the deployment manifest also contains an update location (a Web page or network file share) where the application checks for updated versions. ClickOnce **Publish** properties are used to specify when and how often the application should check for updates. Update behavior can be specified in the deployment manifest, or it can be presented as user choices in the application's user interface by means of the ClickOnce APIs. In addition, **Publish** properties can be employed to make updates mandatory or to roll back to an earlier version. For more information, see [Choosing a ClickOnce update strategy](#).

### Third party installers

You can customize your ClickOnce installer to install third-party components along with your application. You must have the redistributable package (.exe or .msi file) and describe the package with a language-neutral product manifest and a language-specific package manifest. For more information, see [Creating bootstrapper packages](#).

## ClickOnce tools

The following table shows the tools that you can use to generate, edit, sign, and re-sign the application and deployment manifests.

| TOOL  | DESCRIPTION   |
|---|---|
| <a href="#">Security Page, Project Designer</a> | Signs the application and deployment manifests.   |
| <a href="#">Publish Page, Project Designer</a>  | Generates and edits the application and deployment manifests for Visual Basic and Visual C# applications. |

| TOOL  | DESCRIPTION  |
|---|--|
| <a href="#">Mage.exe</a> (Manifest Generation and Editing Tool)                     | <p>Generates the application and deployment manifests for Visual Basic, Visual C#, and Visual C++ applications.</p> <p>Signs and re-signs the application and deployment manifests.</p> <p>Can be run from batch scripts and the command prompt.</p> |
| <a href="#">MageUI.exe</a> (Manifest Generation and Editing Tool, Graphical Client) | <p>Generates and edits the application and deployment manifests.</p> <p>Signs and re-signs the application and deployment manifests.</p>   |
| <a href="#">GenerateApplicationManifest</a> task                                    | <p>Generates the application manifest.</p> <p>Can be run from MSBuild. For more information, see <a href="#">MSBuild reference</a>.</p>  |
| <a href="#">GenerateDeploymentManifest</a> task                                     | <p>Generates the deployment manifest.</p> <p>Can be run from MSBuild. For more information, see <a href="#">MSBuild reference</a>.</p>   |
| <a href="#">SignFile</a> task   | <p>Signs the application and deployment manifests.</p> <p>Can be run from MSBuild. For more information, see <a href="#">MSBuild reference</a>.</p>  |
| <a href="#">Microsoft.Build.Tasks.Deployment.ManifestUtilities</a>                  | <p>Develop your own application to generate the application and deployment manifests.</p>  |

The following table shows the .NET Framework version required to support ClickOnce applications in these browsers.

| BROWSER           | .NET FRAMEWORK VERSION    |
|-------------------|---------------------------|
| Internet Explorer | 2.0, 3.0, 3.5, 3.5 SP1, 4 |
| Firefox           | 2.0 SP1, 3.5 SP1, 4       |
| Chrome            | 3.5                       |
| Microsoft Edge    | 3.5                       |

## See also

- [ClickOnce deployment on Windows Vista](#)
- [Publish ClickOnce applications](#)
- [Secure ClickOnce applications](#)
- [Deploy COM components with ClickOnce](#)
- [Build ClickOnce applications from the command line](#)
- [Debug ClickOnce applications that use System.Deployment.Application](#)

# Choose a ClickOnce deployment strategy

3/5/2021 • 3 minutes to read • [Edit Online](#)

There are three different strategies for deploying a ClickOnce application; the strategy that you choose depends primarily on the type of application that you are deploying. The three deployment strategies are as follows:

- Install from the Web or a Network Share
- Install from a CD
- Start the application from the Web or a Network Share

## NOTE

In addition to selecting a deployment strategy, you will also want to select a strategy for providing application updates. For more information, see [Choose a ClickOnce update strategy](#).

## Install from the Web or a network share

When you use this strategy, your application is deployed to a Web server or a network file share. When an end user wants to install the application, he or she clicks an icon on a Web page or double-clicks an icon on the file share. The application is then downloaded, installed, and started on the end user's computer. Items are added to the **Start** menu and **Add or Remove Programs** in **Control Panel**.

Because this strategy depends on network connectivity, it works best for applications that will be deployed to users who have access to a local-area network or a high-speed Internet connection.

If you deploy the application from the Web, you can pass arguments into the application when it is activated using a URL. For more information, see [How to: Retrieve query string information in an online ClickOnce application](#). You cannot pass arguments into an application that is activated by using any of the other methods described in this document.

To enable this deployment strategy in Visual Studio, click **From the Web** or **From a UNC path or file share** on the **How Installed** page of the Publish Wizard.

This is the default deployment strategy.

## Start the application from the Web or a network share

This strategy is like the first, except the application behaves like a Web application. When the user clicks a link on a Web page (or double-clicks an icon on the file share), the application is started. When users close the application, it is no longer available on their local computer; nothing is added to the **Start** menu or **Add or Remove Programs** in **Control Panel**.

## NOTE

Technically, the application is downloaded and installed to an application cache on the local computer, just as a Web application is downloaded to the Web cache. As with the Web cache, the files are eventually scavenged from the application cache. However, the perception of the user is that the application is being run from the Web or file share.

This strategy works best for applications that are used infrequently—for example, an employee-benefits tool

that is typically run only one time each year.

To enable this deployment strategy in Visual Studio, click **Do not install the application** on the **Install or Run From Web** page of the Publish Wizard.

To enable this deployment strategy, manually, change the **install** tag in the deployment manifest. (Its value can be **true** or **false**. In *Mage.exe*, use the **Online Only** option in the **Application Type** list.)

## Install from a CD

When you use this strategy, your application is deployed to removable media such as a CD-ROM or DVD. As with the previous option, when the user chooses to install the application, it is installed and started, and items are added to the **Start** menu and **Add or Remove Programs** in **Control Panel**.

This strategy works best for applications that will be deployed to users without persistent network connectivity or with low-bandwidth connections. Because the application is installed from removable media, no network connection is necessary for installation; however, network connectivity is still required for application updates.

To enable this deployment strategy in Visual Studio, click **From a CD-ROM or DVD-ROM** on the **How Installed** page of the Publish Wizard.

To enable this deployment strategy manually, change the **deploymentProvider** tag in the deployment manifest. (In Visual Studio, this property is exposed as **Installation URL** on the **Publish** page of the Project Designer. In *Mage.exe* it is **Start Location**.)

## Web browser support

Applications that target .NET Framework 3.5 can be installed using any browser.

Applications that target .NET Framework 2.0 require Internet Explorer.

## See also

- [ClickOnce security and deployment](#)
- [Choose a ClickOnce update strategy](#)
- [How to: Publish a ClickOnce application with the Publish Wizard](#)
- [Securing ClickOnce applications](#)

# ClickOnce cache overview

3/5/2021 • 2 minutes to read • [Edit Online](#)

All ClickOnce applications, whether they are installed locally or hosted online, are stored on the client computer in a ClickOnce application *cache*. A ClickOnce cache is a family of hidden directories under the Local Settings directory of the current user's Documents and Settings folder. This cache holds all the application's files, including the assemblies, configuration files, application and user settings, and data directory. The cache is also responsible for migrating the application's data directory to the latest version. For more information about data migration, see [Accessing Local and Remote Data in ClickOnce Applications](#).

By providing a single location for application storage, ClickOnce takes over the task of managing the physical installation of an application from the user. The cache also helps isolate applications by keeping the assemblies and data files for all applications and their distinct versions separate from one another. For example, when you upgrade a ClickOnce application, that version and its data resources are supplied with their own directories in the cache.

## Cache storage quota

ClickOnce applications that are hosted online are restricted in the amount of space they can occupy by a quota that constrains the size of the ClickOnce cache. The cache size applies to all the user's online applications; a single partially-trusted, online application is limited to occupying half of the quota space. Installed applications are not limited by the cache size and do not count against the cache limit. For all ClickOnce applications, the cache retains only the current version and the previously installed version.

By default, client computers have 250 MB of storage for online ClickOnce applications. Data files do not count toward this limit. A system administrator can enlarge or reduce this quota on a particular client computer by changing the registry key,

**HKEY\_CURRENT\_USER\Software\Classes\Software\Microsoft\Windows\CurrentVersion\Deployment\OnlineAppQuotaInKB**, which is a DWORD value that expresses the cache size in kilobytes. For example, in order to reduce the cache size to 50 MB, you would change this value to 51200.

## See also

- [Access local and remote data in ClickOnce applications](#)

# ClickOnce and application settings

3/5/2021 • 2 minutes to read • [Edit Online](#)

Application settings for Windows Forms makes it easy to create, store, and maintain custom application and user preferences on the client. The following document describes how application settings files work in a ClickOnce application, and how ClickOnce migrates settings when the user upgrades to the next version.

The information below applies only to the default application settings provider, the [LocalFileSettingsProvider](#) class. If you supply a custom provider, that provider will determine how it stores its data and how it upgrades its settings between versions. For more information on application settings providers, see [Application settings architecture](#).

## Application settings files

Application settings consumes two files: *<app>.exe.config* and *user.config*, where *app* is the name of your Windows Forms application. *user.config* is created on the client the first time your application stores user-scoped settings. *<app>.exe.config*, by contrast, will exist prior to deployment if you define default values for settings. Visual Studio will include this file automatically when you use its **Publish** command. If you create your ClickOnce application using *Mage.exe* or *MageUI.exe*, you must make sure this file is included with your application's other files when you populate your application manifest.

In a Windows Forms application not deployed using ClickOnce, an application's *<app>.exe.config* file is stored in the application directory, while the *user.config* file is stored in the user's **Documents and Settings** folder. In a ClickOnce application, *<app>.exe.config* lives in the application directory inside of the ClickOnce application cache, and *user.config* lives in the ClickOnce data directory for that application.

Regardless of how you deploy your application, application settings ensures safe read access to *<app>.exe.config*, and safe read/write access to *user.config*.

In a ClickOnce application, the size of the configuration files used by application settings is constrained by the size of the ClickOnce cache. For more information, see [ClickOnce cache overview](#).

## Version upgrades

Just as each version of a ClickOnce application is isolated from all other versions, the application settings for a ClickOnce application are isolated from the settings for other versions as well. When your user upgrades to a later version of your application, application settings compares most recent (highest-numbered) version's settings against the settings supplied with the updated version and merges the settings into a new set of settings files.

The following table describes how application settings decides which settings to copy.

| TYPE OF CHANGE                                     | UPGRADE ACTION  |
|--|---|
| Setting added to <i>&lt;app&gt;.exe.config</i>     | The new setting is merged into the current version's <i>&lt;app&gt;.exe.config</i>  |
| Setting removed from <i>&lt;app&gt;.exe.config</i> | The old setting is removed from the current version's <i>&lt;app&gt;.exe.config</i> |

| TYPE OF CHANGE   | UPGRADE ACTION  |
|--|---|
| Setting's default changed; local setting still set to original default in <i>user.config</i> | The setting is merged into the current version's <i>user.config</i> with the new default as the value   |
| Setting's default changed; setting set to non-default in <i>user.config</i>                  | The setting is merged into the current version's <i>user.config</i> with the non-default value retained |

If you have created your own application settings wrapper class and wish to customize the update logic, you can override the [Upgrade](#) method.

## ClickOnce and roaming settings

ClickOnce does not work with roaming settings, which allows your settings file to follow you across machines on a network. If you need roaming settings, you will need either to implement an application settings provider that stores settings over the network, or develop your own custom settings classes for storing settings on a remote computer. For more information in settings providers, see [Application settings architecture](#).

## See also

- [ClickOnce security and deployment](#)
- [Application settings overview](#)
- [ClickOnce cache overview](#)
- [Access local and remote data in ClickOnce applications](#)

# ClickOnce deployment on Windows Vista

3/5/2021 • 2 minutes to read • [Edit Online](#)

Building applications in Visual Studio for User Account Control (UAC) on Windows Vista normally generates an embedded manifest, encoded as binary XML data in the application's executable file. ClickOnce and Registration-Free COM applications require an external manifest, so Visual Studio generates a file for these projects containing the UAC data instead of an embedded manifest. For ClickOnce and Registration-Free COM deployments, Visual Studio uses information from a file called *app.manifest* to generate external UAC manifest information. For all other cases, Visual Studio embeds the UAC data in the application's executable file.

Visual Studio provides the following options for manifest generation:

- Use an embedded manifest. Embed UAC data in the application's executable file and run as a normal user.

This is the default setting (unless you use ClickOnce). This setting supports the usual manner in which Visual Studio operates on Windows Vista, with the generation of both an internal and an external manifest using `AsInvoker`.

- Use an external manifest. Generate an external manifest by using *app.manifest*.

This generates only the external manifest by using the information in *app.manifest*. When you publish an application by using ClickOnce or Registration-Free COM, Visual Studio adds *app.manifest* to the project and then adds this option.

- Use no manifest. Create the application without a manifest.

This approach is also known as *virtualization*. Use this option for compatibility with existing applications from earlier versions of Visual Studio.

The new properties are available on the **Application** page of the Project Designer (for Visual C# projects only) and in the MSBuild project file format.

The method for configuring UAC manifest generation in the Visual Studio IDE differs depending on the project type (Visual C# or Visual Basic).

- For information about configuring Visual C# projects for manifest generation, see [Application Page, Project Designer \(C#\)](#).
- For information about configuring Visual Basic projects for manifest generation, see [Application Page, Project Designer \(Visual Basic\)](#).

## See also

- [ClickOnce security and deployment](#)
- [User permissions and Visual Studio](#)
- [Application Page, Project Designer \(C#\)](#)
- [Application Page, Project Designer \(Visual Basic\)](#)



# Localize ClickOnce applications

3/5/2021 • 4 minutes to read • [Edit Online](#)

Localization is the process of making your application appropriate for a specific culture. This process involves translating user interface (UI) text to a region-specific language, using correct date and currency formatting, adjusting the size of controls on a form, and mirroring controls from right to left if necessary.

Localizing your application results in the creation of one or more satellite assemblies. Each assembly contains UI strings, images, and other resources specific to a given culture. (Your application's main executable file contains the strings for the default culture for your application.)

This topic describes three ways to deploy a ClickOnce application for other cultures:

- Include all satellite assemblies in a single deployment.
- Generate one deployment for each culture, with a single satellite assembly included in each.
- Download satellite assemblies on demand.

## Including All Satellite Assemblies in a Deployment

Instead of publishing multiple ClickOnce deployments, you can publish a single ClickOnce deployment that contains all of the satellite assemblies.

This method is the default in Visual Studio. To use this method in Visual Studio, you do not have to do any additional work.

To use this method with *MageUI.exe*, you must set the culture for your application to **neutral** in *MageUI.exe*. Next, you must manually include all of the satellite assemblies in your deployment. In *MageUI.exe*, you can add the satellite assemblies by using the **Populate** button on the **Files** tab of your application manifest.

The benefit of this approach is that it creates a single deployment and simplifies your localized deployment story. At run time, the appropriate satellite assembly will be used, depending on the default culture of the user's Windows operating system. A drawback of this approach is that it downloads all satellite assemblies whenever the application is installed or updated on a client computer. If your application has a large number of strings, or your customers have a slow network connection, this process can affect performance during application update.

### NOTE

This approach assumes that your application adjusts the height, width, and position of controls automatically to accommodate different text string sizes in different cultures. Windows Forms contains a variety of controls and technologies that enable you to design your form to make it easily localizable, including the [FlowLayoutPanel](#) and [TableLayoutPanel](#) controls as well as the [AutoSize](#) property. Also see [How to: Support localization on Windows forms using AutoSize and the TableLayoutPanel control](#).

## Generate one deployment for each culture

In this deployment strategy, you generate multiple deployments. In each deployment, you include only the satellite assembly needed for a specific culture, and you mark the deployment as specific to that culture.

To use this method in Visual Studio, set the **Publish Language** property on the **Publish** tab to the desired region. Visual Studio will automatically include the satellite assembly required for the region you select, and will exclude all other satellite assemblies from the deployment.

You can accomplish the same thing by using the *MageUI.exe* tool in the Microsoft Windows Software Development Kit (SDK). Use the **Populate** button on the **Files** tab of your application manifest to exclude all other satellite assemblies from the application directory, and then set the **Culture** field on the **Name** tab for your deployment manifest in *MageUI.exe*. These steps not only include the correct satellite assembly, but they also set the `language` attribute on the `assemblyIdentity` element in your deployment manifest to the corresponding culture.

After publishing the application, you must repeat this step for each additional culture your application supports. You must make sure that you publish to a different Web server directory or file share directory every time, because each application manifest will reference a different satellite assembly, and each deployment manifest will have a different value for the `language` attribute.

## Download satellite assemblies on demand

If you decide to include all satellite assemblies in a single deployment, you can improve performance by using on-demand downloading, which enables you to mark assemblies as optional. The marked assemblies will not be downloaded when the application is installed or updated. You can install the assemblies when you need them by calling the `DownloadFileGroup` method on the `ApplicationDeployment` class.

Downloading satellite assemblies on demand differs slightly from downloading other types of assemblies on demand. For more information and code examples on how to enable this scenario using the Windows SDK tools for ClickOnce, see [Walkthrough: Downloading Satellite Assemblies on Demand with the ClickOnce Deployment API](#).

You can also enable this scenario in Visual Studio. Also see [Walkthrough: Downloading Satellite Assemblies on Demand with the ClickOnce Deployment API Using the Designer](#) or [Walkthrough: Downloading Satellite Assemblies on Demand with the ClickOnce Deployment API Using the Designer](#).

## Testing localized ClickOnce applications before deployment

A satellite assembly will be used for a Windows Forms application only if the `CurrentUICulture` property for the main thread of the application is set to the satellite assembly's culture. Customers in local markets will probably already be running a localized version of Windows with their culture set to the appropriate default.

You have three options for testing localized deployments before you make your application available to customers:

- You can run your ClickOnce application on the appropriate localized versions of Windows.
- You can set the `CurrentUICulture` property programmatically in your application. (This property must be set before you call the `Run` method.)

## See also

- [<assemblyIdentity> element](#)
- [ClickOnce security and deployment](#)
- [Globalize Windows forms](#)

# How to: Publish a project that has a specific locale

3/5/2021 • 4 minutes to read • [Edit Online](#)

It is not uncommon for an application to contain components that have different locales. In this scenario, you would create a solution that has several projects, and then publish separate projects for each locale. This procedure shows how to use a macro to publish the first project in a solution by using the 'en' locale. If you want to try this procedure with a locale other than 'en', make sure to set `localeString` in the macro to match the locale that you are using (for example, 'de' or 'de-DE').

## NOTE

When you use this macro, the Publish Location should be a valid URL or Universal Naming Convention (UNC) share. Also, Internet Information Services (IIS) has to be installed on your computer. To install IIS, on the **Start** menu, click **Control Panel**. Double-click **Add or Remove Programs**. In **Add or Remove Programs**, click **Add/Remove Windows Components**. In the **Windows Components Wizard**, select the **Internet Information Services (IIS)** check box in the **Components** list. Then click **Finish** to close the wizard.

## To create the publishing macro

1. To open the Macro Explorer, on the **Tools** menu, point to **Macros**, and then click **Macro Explorer**.
2. Create a new macro module. In the Macro Explorer, select **MyMacros**. On the **Tools** menu, point to **Macros**, and then click **New Macro Module**. Name the module **PublishSpecificCulture**.
3. In the Macro Explorer, expand the **MyMacros** node, and then open the **PublishAllProjects** module by double-clicking it (or, from the **Tools** menu, point to **Macros**, and then click **Macros IDE**).
4. In the Macros IDE, add the following code to the module, after the `Import` statements:

```
Module PublishSpecificCulture
    Sub PublishProjectFirstProjectWithEnLocale()
        ' Note: You should publish projects by using the IDE at least once
        ' before you use this macro. Items such as the certificate and the
        ' security zone must be set.
        Dim localeString As String = "en"

        ' Get first project.
        Dim proj As Project = DTE.Solution.Projects.Item(1)
        Dim publishProperties As Object = proj.Properties.Item("Publish").Value

        ' GenerateManifests and SignManifests must always be set to
        ' True for publishing to work.
        proj.Properties.Item("GenerateManifests").Value = True
        proj.Properties.Item("SignManifests").Value = True

        'Set the publish language.
        'This will set the deployment language and pick up all
        ' en resource dlls:
        Dim originalTargetCulture As String = _
            publishProperties.Item("TargetCulture").Value
        publishProperties.Item("TargetCulture").Value = localeString

        'Append 'en' to end of publish, install, and update URLs if needed:
        Dim originalPublishUrl As String = _
            publishProperties.Item("PublishUrl").Value
        Dim originalInstallUrl As String = _
            publishProperties.Item("InstallUrl").Value
        Dim originalUpdateUrl As String = _
```

```

Dim originalUpdateUrl As String = _
    publishProperties.Item("UpdateUrl").Value
publishProperties.Item("PublishUrl").Value = _
    AppendStringToUrl(localeString, New Uri(originalPublishUrl))
If originalInstallUrl <> String.Empty Then
    publishProperties.Item("InstallUrl").Value = _
        AppendStringToUrl(localeString, New Uri(originalInstallUrl))
End If
If originalUpdateUrl <> String.Empty Then
    publishProperties.Item("UpdateUrl").Value = _
        AppendStringToUrl(localeString, New Uri(originalUpdateUrl))
End If
proj.Save()

Dim slnbld2 As SolutionBuild2 = _
    CType(DTE.Solution.SolutionBuild, SolutionBuild2)
slnbld2.Clean(True)

slnbld2.BuildProject( _
    proj.ConfigurationManager.ActiveConfiguration.ConfigurationName, _
    proj.UniqueName, True)

' Only publish if build is successful.
If slnbld2.LastBuildInfo <> 0 Then
    MsgBox("Build failed for " & proj.UniqueName)
Else
    slnbld2.PublishProject( _
        proj.ConfigurationManager.ActiveConfiguration.ConfigurationName, _
        proj.UniqueName, True)
    If slnbld2.LastPublishInfo = 0 Then
        MsgBox("Publish succeeded for " & proj.UniqueName _
            & vbCrLf & "." _
            & " Publish Language was '" & localeString & "'")
    Else
        MsgBox("Publish failed for " & proj.UniqueName)
    End If
End If

' Return URLs and target culture to their previous state.
publishProperties.Item("PublishUrl").Value = originalPublishUrl
publishProperties.Item("InstallUrl").Value = originalInstallUrl
publishProperties.Item("UpdateUrl").Value = originalUpdateUrl
publishProperties.Item("TargetCulture").Value = originalTargetCulture
proj.Save()
End Sub

Private Function AppendStringToUrl(ByVal str As String, _
    ByVal baseUri As Uri) As String
    Dim returnValue As String = baseUri.OriginalString
    If baseUri.IsFile OrElse baseUri.IsUnc Then
        returnValue = IO.Path.Combine(baseUri.OriginalString, str)
    Else
        If Not baseUri.ToString.EndsWith("/") Then
            returnValue = baseUri.OriginalString & "/" & str
        Else
            returnValue = baseUri.OriginalString & str
        End If
    End If
    Return returnValue
End Function
End Module

```

5. Close the Macros IDE. The focus will return to Visual Studio.

### To publish a project for a specific locale

1. To create a Visual Basic Windows Application project, on the **File** menu, point to **New**, and then click **Project**.

2. In the **New Project** dialog box, select **Windows Application** from the **Visual Basic** node. Name the project *PublishLocales*.
3. Click Form1. In the **Properties** window, under **Design**, change the **Language** property from **(Default)** to **English**. Change the **Text** property of the form to **MyForm**.

Note that the localized resource DLLs are not created until they are needed. For example, they are created when you change the text of the form or one of its controls after you have specified the new locale.

4. Publish *PublishLocales* by using the Visual Studio IDE.

In **Solution Explorer**, select *PublishLocales*. On the **Project** menu, select **Properties**. In the Project Designer, on the **Publish** page, specify a publishing location of **http://localhost/PublishLocales**, and then click **Publish Now**.

When the publish Web page appears, close it. (For this step, you only have to publish the project; you do not have to install it.)

5. Publish *PublishLocales* again by invoking the macro in the Visual Studio Command Prompt window. To view the Command Prompt window, on the **View** menu, point to **Other Windows** and then click **Command Window**, or press **Ctrl+Alt+A**. In the Command Prompt window, type `macros`; auto-complete will provide a list of available macros. Select the following macro and press ENTER:

```
Macros.MyMacros.PublishSpecificCulture.PublishProjectFirstProjectWithEnLocale
```

6. When the publish process succeeds, it will generate a message that says "Publish succeeded for *PublishLocales\PublishLocales.vbproj*. Publish language was 'en'." Click **OK** in the message box. When the publish Web page appears, click **Install**.
7. Look in *C:\inetpub\wwwroot\PublishLocales\en*. You should see the installed files such as the manifests, *setup.exe*, and the publish Web page file, in addition to the localized resource DLL. (By default ClickOnce appends a *.deploy* extension on EXEs and DLLs; you can remove this extension after deployment.)

## See also

- [Publish ClickOnce applications](#)
- [Macros development environment](#)
- [Macro Explorer window](#)
- [How to: Edit and programmatically create macros](#)

# Secure ClickOnce applications

3/5/2021 • 5 minutes to read • [Edit Online](#)

ClickOnce applications are subject to code access security constraints in the .NET Framework to help limit the access that code has to protected resources and operations. For that reason, it is important that you understand the implications of code access security to write your ClickOnce applications accordingly. Your applications can use Full Trust or use partial zones, such as the Internet and Intranet zones, to limit access.

Additionally, ClickOnce uses certificates to verify the authenticity of the application's publisher, and to sign the application and deployment manifests to prove that the files have not been tampered with. Signing is an optional step, which makes it easier to change the application files after the manifests are generated. However, without signed manifests, it is difficult to ensure that the application installer is not tampered in man-in-the-middle security attacks. For this reason, we recommend that you sign your application and deployment manifests to help secure your applications.

## Zones

Applications that are deployed using ClickOnce technology are restricted to a set of permissions and actions that are defined by the security zone. Security zones are defined in Internet Explorer, and are based on the location of the application. The following table lists the default permissions based on the deployment location:

| DEPLOYMENT LOCATION             | SECURITY ZONE       |
|---------------------------------|---------------------|
| Run from Web                    | Internet Zone       |
| Install from Web                | Internet Zone       |
| Install from network file share | Local Intranet Zone |
| Install from CD-ROM             | Full Trust          |

The default permissions are based on the location from which the original version of the application was deployed; updates to the application will inherit those permissions. If the application is configured to check for updates from a Web or network location and a newer version is available, the original installation can receive permissions for the Internet or Intranet zone instead of full-trust permissions. To prevent users from being prompted, a system administrator can specify a ClickOnce deployment policy that defines a specific application publisher as a trusted source. For computers on which this policy is deployed, permissions will be granted automatically and the user will not be prompted. For more information, see [Trusted Application Deployment Overview](#). To configure trusted application deployment, the certificate can be installed to the machine or enterprise level. For more information, see [How to: Add a Trusted Publisher to a Client Computer for ClickOnce Applications](#).

## Code access security policies

Permissions for an application are determined by the settings in the `<trustInfo>` Element element of the application manifest. Visual Studio automatically generates this information based on the settings on the project's **Security** property page. A ClickOnce application is granted only the specific permissions that it requests. For example, where file access requires full-trust permissions, if the application requests file-access permission, it will only be granted file-access permission, not full-trust permissions. When developing your ClickOnce application, you should make sure that you request only the specific permissions that the application

needs. In most cases, you can use the Internet or Local Intranet zones to limit your application to partial trust. For more information, see [How to: Set a security zone for a ClickOnce application](#). If your application requires custom permissions, you can create a custom zone. For more information, see [How to: Set custom permissions for a ClickOnce application](#).

Including a permission that is not part of the default permission set for the zone from which the application is deployed will cause the end user to be prompted to grant permission at install or update time. To prevent users from being prompted, a system administrator can specify a ClickOnce deployment policy that defines a specific application publisher as a trusted source. On computers where this policy is deployed, permissions will automatically be granted and the user will not be prompted.

As a developer, it is your responsibility to make sure that your application will run with the appropriate permissions. If the application requests permissions outside of a zone during run time, a security exception may appear. Visual Studio enables you to debug your application in the target security zone and provides help in developing secure applications. For more information, see [Debug ClickOnce apps that use System.Deployment.Application](#).

For more information about code access security and ClickOnce, see [Code access security for ClickOnce applications](#).

## Code-signing certificates

To publish an application by using ClickOnce deployment, you can sign the application and deployment manifests for the application by using a public/private key pair. The tools for signing a manifest are available on the **Signing** page of the **Project Designer**. For more information, see [Signing Page, Project Designer](#).

After the manifests are signed, the publisher information based on the Authenticode signature will be displayed to the user in the permissions dialog box during installation, to show the user that the application originated from a trusted source.

For more information about ClickOnce and certificates, see [ClickOnce and Authenticode](#).

## ASP.NET form-based authentication

If you want to control which deployments each user can access, you should not enable anonymous access to ClickOnce applications deployed on a Web server. Rather, you would enable users access to the deployments you have installed based on a user's identity using Windows authentication.

ClickOnce does not support ASP.NET forms-based authentication because it uses persistent cookies; these present a security risk because they reside in the Internet Explorer cache and can be hacked. Therefore, if you are deploying ClickOnce applications, any authentication scenario besides Windows authentication is unsupported.

## Pass arguments

An additional security consideration occurs if you have to pass arguments into a ClickOnce application. ClickOnce enables developers to supply a query string to applications deployed over the Web. The query string takes the form of a series of name-value pairs at the end of the URL used to start the application:

```
http://servername.adatum.com/WindowsApp1.application?username=joeuser
```

By default, query-string arguments are disabled. To enable them, the attribute `trustUrlParameters` must be set in the application's deployment manifest. This value can be set from Visual Studio and from MageUI.exe. For detailed steps on how to enable passing query strings, see [How to: Retrieve query string information in an online ClickOnce application](#).

You should never pass arguments retrieved through a query string to a database or to the command line

without checking the arguments to make sure that they are safe. Unsafe arguments are ones that include database or command line escape characters that could allow a malicious user to manipulate your application into executing arbitrary commands.

#### NOTE

Query-string arguments are the only way to pass arguments to a ClickOnce application at startup. You cannot pass arguments to a ClickOnce application from the command line.

## Deploying obfuscated assemblies

Visual Studio includes the free [PreEmptive Protection - Dotfuscator Community](#), which you can use to protect your ClickOnce applications through code obfuscation and active protection measures. For details, please see [the ClickOnce section of the Dotfuscator Community User Guide](#).

## See also

- [ClickOnce security and deployment](#)
- [Choose a ClickOnce deployment strategy](#)



# ClickOnce and Authenticode

3/5/2021 • 3 minutes to read • [Edit Online](#)

*Authenticode* is a Microsoft technology that uses industry-standard cryptography to sign application code with digital certificates that verify the authenticity of the application's publisher. By using Authenticode for application deployment, ClickOnce reduces the risk of a Trojan horse. A Trojan horse is a virus or other harmful program that a malicious third party misrepresents as a legitimate program coming from an established, trustworthy source. Signing ClickOnce deployments with a digital certificate is an optional step to verify that the assemblies and files are not tampered.

The following sections describe the different types of digital certificates used in Authenticode, how certificates are validated using Certificate Authorities (CAs), the role of time-stamping in certificates, and the methods of storage available for certificates.

## Authenticode and code signing

A *digital certificate* is a file that contains a cryptographic public/private key pair, along with metadata describing the publisher to whom the certificate was issued and the agency that issued the certificate.

There are various types of Authenticode certificates. Each one is configured for different types of signing. For ClickOnce applications, you must have an Authenticode certificate that is valid for code signing. If you attempt to sign a ClickOnce application with another type of certificate, such as a digital e-mail certificate, it will not work. For more information, see [Introduction to code signing](#).

You can obtain a certificate for code signing in one of three ways:

- Purchase one from a certificate vendor.
- Receive one from a group in your organization responsible for creating digital certificates.
- Generate your own certificate by using the New-SelfSignedCertificate PowerShell cmdlet, or by using *MakeCert.exe*, which is included with the Windows Software Development Kit (SDK).

### How using certificate authorities helps users

A certificate generated using New-SelfSignedCertificate or the *MakeCert.exe* utility is commonly called a *self-cert* or a *test cert*. This kind of certificate works much the same way that a *.snk* file works in the .NET Framework. It consists solely of a public/private cryptographic key pair, and contains no verifiable information about the publisher. You can use self-certs to deploy ClickOnce applications with high trust on an intranet. However, when these applications run on a client computer, ClickOnce will identify them as coming from an Unknown Publisher. By default, ClickOnce applications signed with self-certs and deployed over the Internet cannot utilize Trusted Application Deployment.

By contrast, if you receive a certificate from a CA, such as a certificate vendor, or a department within your enterprise, the certificate offers more security for your users. It not only identifies the publisher of the signed software, but it verifies that identity by checking with the CA that signed it. If the CA is not the root authority, Authenticode will also "chain" back to the root authority to verify that the CA is authorized to issue certificates. For greater security, you should use a certificate issued by a CA whenever possible.

For more information about generating self-certs, see [New-SelfSignedCertificate](#) or [MakeCert](#).

### Timestamps

The certificates used to sign ClickOnce applications expire after a certain length of time, typically twelve months. In order to remove the need to constantly re-sign applications with new certificates, ClickOnce supports

timestamp. When an application is signed with a timestamp, its certificate will continue to be accepted even after expiration, provided the timestamp is valid. This allows ClickOnce applications with expired certificates, but valid timestamps, to download and run. It also allows installed applications with expired certificates to continue to download and install updates.

To include a timestamp in an application server, a timestamp server must be available. For information about how to select a timestamp server, see [How to: Sign Application and Deployment Manifests](#).

### Update expired certificates

In earlier versions of the .NET Framework, updating an application whose certificate had expired could cause that application to stop functioning. To resolve this problem, use one of the following methods:

- Update the .NET Framework to version 2.0 SP1 or later on Windows XP, or version 3.5 or later on Windows Vista.
- Uninstall the application, and reinstall a new version with a valid certificate.

### Store certificates

- You can store certificates as a *.pfx* file on your file system, or you can store them inside of a key container. A user on a Windows domain can have a number of key containers. By default, *MakeCert.exe* will store certificates in your personal key container, unless you specify that it should save it to a *.pfx* instead. *Mage.exe* and *MageUI.exe*, the Windows SDK tools for creating ClickOnce deployments, enable you to use certificates stored in either fashion.

## See also

- [ClickOnce security and deployment](#)
- [Secure ClickOnce applications](#)
- [Trusted application deployment overview](#)
- [Mage.exe \(Manifest Generation and Editing Tool\)](#)

# Trusted Application Deployment overview

3/5/2021 • 5 minutes to read • [Edit Online](#)

This topic provides an overview of how to deploy ClickOnce applications that have elevated permissions by using the Trusted Application Deployment technology.

Trusted Application Deployment, part of the ClickOnce deployment technology, makes it easier for organizations of any size to grant additional permissions to a managed application in a safer, more secure manner without user prompting. With Trusted Application Deployment, an organization can just configure a client computer to have a list of trusted publishers, who are identified using Authenticode certificates. Thereafter, any ClickOnce application signed by one of these trusted publishers receives a higher level of trust.

## NOTE

Trusted Application Deployment requires one-time configuration of a user's computer. In managed desktop environments, this configuration can be performed by using global policy. If this is not what you want for your application, use permission elevation instead. For more information, see [Securing ClickOnce Applications](#).

## Trusted Application Deployment basics

The following table shows the objects and roles that are involved in Trusted Application Deployment.

| OBJECT OR ROLE           | DESCRIPTION  |
|--------------------------|--|
| administrator            | The organizational entity responsible for updating and maintaining client computers  |
| trust manager            | The subsystem within the common language runtime (CLR) responsible for enforcing client application security.  |
| publisher                | The entity that writes and maintains the application.  |
| deployer                 | The entity that packages and distributes the application to users.   |
| certificate              | A cryptographic signature that consists of a public and private key; generally issued by a certification authority (CA) that can vouch for its authenticity. |
| Authenticode certificate | A certificate with embedded metadata describing, among other things, the uses for which the certificate can be employed.                                     |
| certification authority  | An organization that verifies the identity of publishers and issues them certificates embedded with the publisher's metadata.                                |
| root authority           | A certification authority that authorizes other Certificate Authorities to issue certificates.   |

| OBJECT OR ROLE    | DESCRIPTION   |
|-------------------|---|
| key container     | A logical storage space in Microsoft Windows for storing certificates.  |
| trusted publisher | A publisher whose Authenticode certificate has been added to a certificate trust list (CTL) on a client computer. |

In larger organizations, the publisher and deployer are frequently two separate entities:

- The publisher is the group that creates the ClickOnce application.
- The deployer is the group, typically the information technology (IT) department, that distributes ClickOnce application to corporate enterprise desktop computers.

You must follow these steps to take advantage of Trusted Application Deployment:

1. Obtain a certificate for the publisher.
2. Add the publisher to the trusted publishers store on all clients.
3. Create your ClickOnce application.
4. Sign the deployment manifest with the publisher's certificate.
5. Publish the application deployment to client computers.

### Obtain a certificate for the publisher

Digital certificates are a core component of the Microsoft Authenticode authentication and security system. Authenticode is a standard part of the Windows operating system. All ClickOnce applications must be signed with a digital certificate, regardless of whether they participate in Trusted Application Deployment. For a full explanation of how Authenticode works with ClickOnce, see [ClickOnce and Authenticode](#).

### Add the publisher to the trusted publishers store

For your ClickOnce application to receive a higher level of trust, you must add your certificate as a trusted publisher to each client computer on which the application will run. Performing this task is a one-time configuration. After it is completed, you can deploy as many ClickOnce applications signed with your publisher's certificate as you want, and they will all run with high trust.

If you are deploying your application in a managed desktop environment; for example, a corporate intranet running the Windows operating system; you can add trusted publishers to a client's store by creating a new certificate trust list (CTL) with Group Policy. For more information, see [Create a certificate trust list for a Group Policy object](#).

If you are not deploying your application in a managed desktop environment, you have the following options for adding a certificate to the trusted publisher store:

- The [System.Security.Cryptography](#) namespace.
- *CertMgr.exe*, which is a component of Internet Explorer and therefore exists on Windows 98 and all later versions. For more information, see [Certmgr.exe \(Certificate Manager Tool\)](#).

### Create a ClickOnce Application

A ClickOnce application is a .NET Framework client application combined with manifest files that describe the application and supply installation parameters. You can turn your program into a ClickOnce application by using the **Publish** command in Visual Studio. Alternatively, you can generate all the files required for ClickOnce deployment by using tools that are included with the Windows Software Development Kit (SDK). For detailed steps about ClickOnce deployment, see [Walkthrough: Manually Deploying a ClickOnce Application](#).

Trusted Application Deployment is specific to ClickOnce, and can only be used with ClickOnce applications.

### Sign the deployment

After obtaining your certificate, you must use it to sign your deployment. If you are deploying your application by using the Visual Studio Publish wizard, the wizard will automatically generate a test certificate for you if you have not specified a certificate yourself. You can also use the Visual Studio Project Designer window, however, to supply a certificate provided by a CA. Also see [How to: Publish a ClickOnce Application using the Publish Wizard](#).

#### Caution

We do not recommend that the application be deployed with a test certificate.

You can also sign the application by using the *Mage.exe* or *MageUI.exe* SDK tools. For more information, see [Walkthrough: Manually deploy a ClickOnce application](#). For a full list of command-line options related to deployment signing, see [Mage.exe \(Manifest Generation and Editing Tool\)](#).

### Publish the application

As soon as you have signed your ClickOnce manifests, the application is ready to publish to your install location. The installation location can be a Web server, a file share, or the local disk. When a client accesses the deployment manifest for the first time, the trust manager must choose whether the ClickOnce application has been granted authority or not to run at a higher level of trust by an installed trusted publisher. The trust manager makes this choice by comparing the certificate used to sign the deployment with the certificates stored in the client's trusted publisher store. If the trust manager finds a match, the application runs with high trust.

## Trusted Application Deployment and Permission Elevation

If the current publisher is not a trusted publisher, trust manager will use Permission Elevation to query the user about whether he or she wants to grant your application elevated permissions. If permission elevation is disabled by the administrator, however, the application cannot obtain permission to run. The application will not run and no notification will be displayed to the user. For more information about Permission Elevation, see [Securing ClickOnce Applications](#).

## Limitations of Trusted Application Deployment

You can use Trusted Application Deployment to grant elevated trust to ClickOnce applications deployed over the Web or through an enterprise file share. You do not have to use Trusted Application Deployment for ClickOnce applications distributed on a CD, because, by default, these applications are granted full trust.

### See also

- [Mage.exe \(Manifest Generation and Editing Tool\)](#)
- [Walkthrough: Manually deploy a ClickOnce application](#)

# Code access security for ClickOnce applications

3/5/2021 • 4 minutes to read • [Edit Online](#)

ClickOnce applications are based on the .NET Framework and are subject to code access security constraints. For this reason, it is important that you understand the implications of code access security and write your ClickOnce applications accordingly.

Code access security is a mechanism in the .NET Framework that helps limit the access that code has to protected resources and operations. You should configure the code access security permissions for your ClickOnce application to use the zone appropriate for the location of the application installer. In most cases, you can choose the **Internet** zone for a limited set of permissions or the **Local Intranet** zone for a greater set of permissions.

## Default ClickOnce code access security

By default, a ClickOnce application receives Full Trust permissions when it is installed or run on a client computer.

- An application that has Full Trust permissions has unrestricted access to resources such as the file system and the registry. This potentially allows your application (and the end user's system) to be exploited by malicious code.
- When an application requires Full Trust permissions, the end user may be prompted to grant permissions to the application. This means that the application does not truly provide a ClickOnce experience, and the prompt can potentially be confusing to less experienced users.

### NOTE

When installing an application from removable media such as a CD-ROM, the user is not prompted. In addition, a network administrator can configure network policy so that users are not prompted when they install an application from a trusted source. For more information, see [Trusted application deployment overview](#).

To restrict the permissions for a ClickOnce application, you can modify the code access security permissions for your application to request the zone that best fits the permissions that your application requires. In most cases, you can select the zone from which the application is being deployed. For example, if your application is an enterprise application, you can use the **Local Intranet** zone. If your application is an internet application, you can use the **Internet** zone.

## Configure security permissions

You should always configure your ClickOnce application to request the appropriate zone to limit the code access security permissions. You can configure security permissions on the **Security** page of the **Project Designer**.

The **Security** page in the **Project Designer** contains an **Enable ClickOnce Security Settings** check box. When this check box is selected, security permission requests are added to the deployment manifest for your application. At installation time, the user will be prompted to grant permissions if the requested permissions exceed the default permissions for the zone from which the application is deployed. For more information, see [How to: Enable ClickOnce security settings](#).

Applications deployed from different locations are granted different levels of permissions without prompting. For example, when an application is deployed from the Internet, it receives a highly restrictive set of

permissions. When installed from a local Intranet, it receives more permissions, and when installed from a CD-ROM, it receives Full Trust permissions.

As a starting point for configuring permissions, you can select a security zone from the **Zone** list on the **Security** page. If your application will potentially be deployed from more than one zone, select the zone with the least permissions. For more information, see [How to: Set a security zone for a ClickOnce application](#).

The properties that can be set vary by permission set; not all permission sets have configurable properties. For more information about the full list of permissions that your application can request, see [System.Security.Permissions](#). For more information about how to set permissions for a custom zone, see [How to: Set custom permissions for a ClickOnce application](#).

## Debug an application that has restricted permissions

As a developer, you most likely run your development computer with Full Trust permissions. Therefore, you do not see the same security exceptions when you debug the application that users may see when they run it with restricted permissions.

In order to catch these exceptions, you have to debug the application with the same permissions as the end user. Debugging with restricted permissions can be enabled on the **Security** page of the **Project Designer**.

When you debug an application with restricted permissions, exceptions will be raised for any code security demands that have not been enabled on the **Security** page. An exception helper will appear, providing suggestions about how to modify your code to prevent the exception.

In addition, when you write code, the IntelliSense feature in the Code Editor will disable any members that are not included in the security permissions that you have configured.

For more information, see [How to: Debug a ClickOnce Application with Restricted Permissions](#).

## Security permissions for browser-hosted applications

Visual Studio provides the following project types for Windows Presentation Foundation (WPF) applications:

- WPF Windows Application
- WPF Web Browser Application
- WPF Custom Control Library
- WPF Service Library

Of these project types, only WPF Web Browser Applications are hosted in a Web browser and therefore require special deployment and security settings. The default security settings for these applications are as follows:

- **Enable ClickOnce Security Settings**
- **This is a partial trust application**
- **Internet zone** (with default permission set for WPF Web Browser Applications selected)

In the **Advanced Security Settings** dialog box, the **Debug this application with the selected permission set** check box is selected and disabled. This is because Debug In Zone cannot be turned off for browser-hosted applications.

## See also

- [Secure ClickOnce applications](#)

- [How to: Enable ClickOnce security settings](#)
- [How to: Set a security zone for a ClickOnce application](#)
- [How to: Set custom permissions for a ClickOnce application](#)
- [How to: Debug a ClickOnce application with restricted permissions](#)
- [Trusted application deployment overview](#)
- [Security Page, Project Designer](#)



# How to: Enable ClickOnce security settings

3/5/2021 • 2 minutes to read • [Edit Online](#)

Code access security for ClickOnce applications must be enabled in order to publish the application. This is done automatically when you publish an application using the Publish wizard.

In some cases, enabling code access security can impact performance when building or debugging your application; in these cases, you may wish to temporarily disable the security settings.

ClickOnce security settings can be enabled or disabled on the **Security** page of the **Project Designer**.

## To enable ClickOnce security settings

1. With a project selected in **Solution Explorer**, on the **Project** menu, click **Properties**.
2. Click the **Security** tab.
3. Select the **Enable ClickOnce Security Settings** check box.

You can now customize the security settings for your application on the Security page.

### NOTE

This check box is automatically selected each time the application is published with the **Publish** wizard.

## To disable ClickOnce security settings

1. With a project selected in **Solution Explorer**, on the **Project** menu, click **Properties**.
2. Click the **Security** tab.
3. Clear the **Enable ClickOnce Security Settings** check box.

Your application will be run with the full trust security settings; any settings on the **Security** page will be ignored.

### NOTE

Each time the application is published with the Publish wizard, this check box will be selected; you must clear it again after each successful publish.

## See also

- [Secure ClickOnce applications](#)
- [Code access security for ClickOnce applications](#)

# How to: Set a security zone for a ClickOnce application

3/5/2021 • 2 minutes to read • [Edit Online](#)

When setting code access security permissions for a ClickOnce application, you need to start with a base set of permissions on the **Security** page of the **Project Designer**.

In most cases, you can also choose the **Internet** zone which contains a limited set of permissions, or the **Local Intranet** zone which contains a greater set of permissions. If your application requires custom permissions, you can do so by choosing the **Custom** security zone. For more information about setting custom permissions, see [How to: Set Custom Permissions for a ClickOnce Application](#).

## To set a security zone

1. With a project selected in **Solution Explorer**, on the **Project** menu click **Properties**.
2. Click the **Security** tab.
3. Select the **Enable ClickOnce Security Settings** check box.
4. Select the **This is a partial trust application** option button.

The controls in the **ClickOnce security permissions** section are enabled.

5. In the **Zone your application will be installed from** drop-down list, select a security zone.

## See also

- [How to: Set custom permissions for a ClickOnce application](#)
- [Secure ClickOnce applications](#)
- [Code access security for ClickOnce applications](#)

# How to: Set custom permissions for a ClickOnce application

3/5/2021 • 2 minutes to read • [Edit Online](#)

You can deploy a ClickOnce application that uses default permissions for the Internet or Local Intranet zones. Alternatively, you can create a custom zone for the specific permissions that the application needs. You can do this by customizing the security permissions on the **Security** page of the **Project Designer**.

## To customize a permission

1. With a project selected in **Solution Explorer**, on the **Project** menu, click **Properties**.
2. Click the **Security** tab.
3. Select the **Enable ClickOnce Security Settings** check box.
4. Select the **This is a partial trust application** option button.

The controls in the **ClickOnce security permissions** section are enabled.

5. From the **Zone your application will be installed from** drop-down list, click **(Custom)**.
6. Click **Edit Permissions XML**.

The *app.manifest* file opens in the XML Editor.

7. Before the `</applicationRequestMinimum>` element, add XML code for permissions that your application requires.

### NOTE

You can use the `ToXml` method of a permission set to generate the XML code for the application manifest. For example, to generate the XML for the `EnvironmentPermission` permission set, call the `ToXml` method.

## See also

- [Secure ClickOnce Applications](#)
- [Code access security for ClickOnce applications](#)

# How to: Add a trusted publisher to a client computer for ClickOnce applications

3/5/2021 • 2 minutes to read • [Edit Online](#)

With Trusted Application Deployment, you can configure client computers so that your ClickOnce applications run with a higher level of trust without prompting the user. The following procedures show how to use the command-line tool CertMgr.exe to add a publisher's certificate to the Trusted Publishers store on a client computer.

The commands you use vary slightly depending on whether the certificate authority (CA) that issued your certificate is part of a client's trusted root. If a Windows client computer is part of a domain, it will contain, in a list, CAs that are considered trusted roots. This list is usually configured by the system administrator. If your certificate was issued by one of these trusted roots, or by a CA that chains to one of these trusted roots, you can add the certificate to the client's trusted root store. If, on the other hand, your certificate was not issued by one of these trusted roots, you must add the certificate to both the client's Trusted Root store and Trusted Publisher store.

## NOTE

You must add certificates this way on every client computer to which you plan to deploy a ClickOnce application that requires elevated permissions. You add the certificates either manually or through an application you deploy to your clients. You only need to configure these computers once, after which you can deploy any number of ClickOnce applications signed with the same certificate.

You may also add a certificate to a store programmatically using the [X509Store](#) class.

For an overview of Trusted Application Deployment, see [Trusted application deployment overview](#).

## To add a certificate to the Trusted Publishers store under the trusted root

1. Obtain a digital certificate from a CA.
2. Export the certificate into the Base64 X.509 (.cer) format. For more information about certificate formats, see [Export a certificate](#).
3. From the command prompt on client computers, run the following command:

```
certmgr.exe -add certificate.cer -c -s -r localMachine TrustedPublisher
```

## To add a certificate to the Trusted Publishers store under a different root

1. Obtain a digital certificate from a CA.
2. Export the certificate into the Base64 X.509 (.cer) format. For more information about certificate formats, see [Export a Certificate](#).
3. From the command prompt on client computers, run the following command:

```
certmgr.exe -add good.cer -c -s -r localMachine Root
```

```
certmgr.exe -add good.cer -c -s -r localMachine TrustedPublisher
```

See also

- [Walkthrough: Manually deploy a ClickOnce application](#)
- [Secure ClickOnce applications](#)
- [Code access security for ClickOnce applications](#)
- [ClickOnce and Authenticode](#)
- [Trusted application deployment overview](#)
- [How to: Enable ClickOnce security settings](#)
- [How to: Set a security zone for a ClickOnce application](#)
- [How to: Set custom permissions for a ClickOnce application](#)
- [How to: Debug a ClickOnce application with restricted permissions](#)
- [How to: Add a trusted publisher to a client computer for ClickOnce applications](#)
- [How to: Re-sign application and deployment manifests](#)
- [How to: Configure the ClickOnce trust prompt behavior](#)

# How to: Re-sign application and deployment manifests

3/5/2021 • 3 minutes to read • [Edit Online](#)

After you make changes to deployment properties in the application manifest for Windows Forms applications, Windows Presentation Foundation applications (xbap), or Office solutions, you must re-sign both the application and deployment manifests with a certificate. This process helps ensure that tampered files are not installed on end user computers.

Another scenario where you might re-sign the manifests is when your customers want to sign the application and deployment manifests with their own certificate.

## Re-sign the Application and Deployment Manifests

This procedure assumes that you have already made changes to your application manifest file (*.manifest*). For more information, see [How to: Change deployment properties](#).

### To re-sign the application and deployment manifests with Mage.exe

1. Open a **Visual Studio Command Prompt** window.
2. Change directories to the folder that contains the manifest files that you want to sign.
3. Type the following command to sign the application manifest file. Replace *ManifestFileName* with the name of your manifest file plus the extension. Replace *Certificate* with the relative or fully qualified path of the certificate file and replace *Password* with the password for the certificate.

```
mage -sign ManifestFileName.manifest -CertFile Certificate -Password Password
```

For example, you could run the following command to sign an application manifest for an add-in, a Windows Form application, or a Windows Presentation Foundation browser application. Temporary certificates created by Visual Studio are not recommended for deployment into production environments.

```
mage -sign WindowsFormsApplication1.exe.manifest -CertFile  
..\WindowsFormsApplication1_TemporaryKey.pfx  
mage -sign ExcelAddin1.dll.manifest -CertFile ..\ExcelAddIn1_TemporaryKey.pfx  
mage -sign WpfBrowserApplication1.exe.manifest -CertFile ..\WpfBrowserApplication1_TemporaryKey.pfx
```

4. Type the following command to update and sign the deployment manifest file, replacing the placeholder names as in the previous step.

```
mage -update DeploymentManifest -appmanifest ApplicationManifest -CertFile Certificate -Password  
Password
```

For example, you could run the following command to update and sign a deployment manifest for an Excel add-in, a Windows Forms application, or a Windows Presentation Foundation browser application.

```
mage -update WindowsFormsApplication1.application -appmanifest WindowsFormsApplication1.exe.manifest
-CertFile ..\WindowsFormsApplication1_TemporaryKey.pfx
mage -update ExcelAddin1.vsto -appmanifest ExcelAddin1.dll.manifest -CertFile
..\ExcelAddin1_TemporaryKey.pfx
mage -update WpfBrowserApplication1.xbap -appmanifest WpfBrowserApplication1.exe.manifest -CertFile
..\WpfBrowserApplication1_TemporaryKey.pfx
```

5. Optionally, copy the master deployment manifest (*publish\<appname>.application*) to your version deployment directory (*publish\Application Files\<appname>\_<version>*).

## Update and re-sign the application and deployment manifests

This procedure assumes that you have already made changes to your application manifest file (*.manifest*), but that there are other files that were updated. When files are updated, the hash that represents the file must also be updated.

**To update and re-sign the application and deployment manifests with Mage.exe**

1. Open a **Visual Studio Command Prompt** window.
2. Change directories to the folder that contains the manifest files that you want to sign.
3. Remove the *.deploy* file extension from the files in the publish output folder.
4. Type the following command to update the application manifest with the new hashes for the updated files and sign the application manifest file. Replace *ManifestFileName* with the name of your manifest file plus the extension. Replace *Certificate* with the relative or fully qualified path of the certificate file and replace *Password* with the password for the certificate.

```
mage -update ManifestFileName.manifest -CertFile Certificate -Password Password
```

For example, you could run the following command to sign an application manifest for an add-in, a Windows Form application, or a Windows Presentation Foundation browser application. Temporary certificates created by Visual Studio are not recommended for deployment into production environments.

```
mage -update WindowsFormsApplication1.exe.manifest -CertFile
..\WindowsFormsApplication1_TemporaryKey.pfx
mage -update ExcelAddin1.dll.manifest -CertFile ..\ExcelAddin1_TemporaryKey.pfx
mage -update WpfBrowserApplication1.exe.manifest -CertFile ..\WpfBrowserApplication1_TemporaryKey.pfx
```

5. Type the following command to update and sign the deployment manifest file, replacing the placeholder names as in the previous step.

```
mage -update DeploymentManifest -appmanifest ApplicationManifest -CertFile Certificate -Password
Password
```

For example, you could run the following command to update and sign a deployment manifest for an Excel add-in, a Windows Forms application, or a Windows Presentation Foundation browser application.

```
mage -update WindowsFormsApplication1.application -appmanifest WindowsFormsApplication1.exe.manifest
-CertFile ..\WindowsFormsApplication1_TemporaryKey.pfx
mage -update ExcelAddin1.vsto -appmanifest ExcelAddin1.dll.manifest -CertFile
..\ExcelAddin1_TemporaryKey.pfx
mage -update WpfBrowserApplication1.xbap -appmanifest WpfBrowserApplication1.exe.manifest -CertFile
..\WpfBrowserApplication1_TemporaryKey.pfx
```

6. Add the *.deploy* file extension back to the files, except the application and deployment manifest files.
7. Optionally, copy the master deployment manifest (*publish\<appname>.application*) to your version deployment directory (*publish\Application Files\<appname>\_<version>*).

## See also

- [Secure ClickOnce applications](#)
- [Code access security for ClickOnce applications](#)
- [ClickOnce and Authenticode](#)
- [Trusted application deployment overview](#)
- [How to: Enable ClickOnce security settings](#)
- [How to: Set a security zone for a ClickOnce application](#)
- [How to: Set custom permissions for a ClickOnce application](#)
- [How to: Debug a ClickOnce application with restricted permissions](#)
- [How to: Add a trusted publisher to a client computer for ClickOnce applications](#)
- [How to: Configure the ClickOnce trust prompt behavior](#)



# How to: Configure the ClickOnce trust prompt behavior

3/5/2021 • 4 minutes to read • [Edit Online](#)

You can configure the ClickOnce trust prompt to control whether end users are given the option of installing ClickOnce applications, such as Windows Forms applications, Windows Presentation Foundation applications, console applications, WPF browser applications, and Office solutions. You configure the trust prompt by setting registry keys on each end user's computer.

The following table shows the configuration options that can be applied to each of the five zones (Internet, UntrustedSites, MyComputer, LocalIntranet, and TrustedSites).

| OPTION                     | REGISTRY SETTING VALUE | DESCRIPTION  |
|----------------------------|------------------------|--|
| Enable the trust prompt.   | Enabled                | The ClickOnce trust prompt is display so that end users can grant trust to ClickOnce applications.                                     |
| Restrict the trust prompt. | AuthenticodeRequired   | The ClickOnce trust prompt is only displayed if ClickOnce applications are signed with a certificate that identifies the publisher.    |
| Disable the trust prompt.  | Disabled               | The ClickOnce trust prompt is not displayed for any ClickOnce applications that are not signed with an explicitly trusted certificate. |

The following table shows the default behavior for each zone. The Applications column refers to Windows Forms applications, Windows Presentation Foundation applications, WPF browser applications, and console applications.

| ZONE           | APPLICATIONS | OFFICE SOLUTIONS     |
|----------------|--------------|----------------------|
| MyComputer     | Enabled      | Enabled              |
| LocalIntranet  | Enabled      | Enabled              |
| TrustedSites   | Enabled      | Enabled              |
| Internet       | Enabled      | AuthenticodeRequired |
| UntrustedSites | Disabled     | Disabled             |

You can override these settings by enabling, restricting, or disabling the ClickOnce trust prompt.

## Enable the ClickOnce trust prompt

Enable the trust prompt for a zone when you want end users to be presented with the option of installing and running any ClickOnce application that comes from that zone.

### To enable the ClickOnce trust prompt by using the registry editor

1. Open the registry editor:
  - a. Click **Start**, and then click **Run**.
  - b. In the **Open** box, type `regedit`, and then click **OK**.

2. Find the following registry key:

**\HKEY\_LOCAL\_MACHINE\SOFTWARE\MICROSOFT\.NETFramework\Security\TrustManager\PromptingLevel**

If the key does not exist, create it.

3. Add the following subkeys as **String Value**, if they do not already exist, with the associated values shown in the following table.

| STRING VALUE SUBKEY | VALUE    |
|---------------------|----------|
| Internet            | Enabled  |
| UntrustedSites      | Disabled |
| MyComputer          | Enabled  |
| LocalIntranet       | Enabled  |
| TrustedSites        | Enabled  |

For Office solutions, `Internet` has the default value `AuthenticodeRequired` and `UntrustedSites` has the value `Disabled`. For all others, `Internet` has the default value `Enabled`.

### To enable the ClickOnce trust prompt programmatically

1. Create a Visual Basic or Visual C# console application in Visual Studio.
2. Open the *Program.vb* or *Program.cs* file for editing and add the following code.

```
Dim key As Microsoft.Win32.RegistryKey
key =
Microsoft.Win32.Registry.LocalMachine.CreateSubKey("SOFTWARE\MICROSOFT\.NETFramework\Security\TrustMa
nager\PromptingLevel")
key.SetValue("MyComputer", "Enabled")
key.SetValue("LocalIntranet", "Enabled")
key.SetValue("Internet", "Enabled")
key.SetValue("TrustedSites", "Enabled")
key.SetValue("UntrustedSites", "Disabled")
key.Close()
```

```
Microsoft.Win32.RegistryKey key;
key =
Microsoft.Win32.Registry.LocalMachine.CreateSubKey("SOFTWARE\\MICROSOFT\\.NETFramework\\Security\\Tru
stManager\\PromptingLevel");
key.SetValue("MyComputer", "Enabled");
key.SetValue("LocalIntranet", "Enabled");
key.SetValue("Internet", "AuthenticodeRequired");
key.SetValue("TrustedSites", "Enabled");
key.SetValue("UntrustedSites", "Disabled");
key.Close();
```

3. Build and run the application.

## Restrict the ClickOnce trust prompt

Restrict the trust prompt so that solutions must be signed with Authenticode certificates that have known identity before users are prompted for a trust decision.

### To restrict the ClickOnce trust prompt by using the registry editor

1. Open the registry editor:
  - a. Click **Start**, and then click **Run**.
  - b. In the **Open** box, type `regedit`, and then click **OK**.
2. Find the following registry key:

**\HKEY\_LOCAL\_MACHINE\SOFTWARE\MICROSOFT\.NETFramework\Security\TrustManager\PromptingLevel**

If the key does not exist, create it.

3. Add the following subkeys as **String Value**, if they do not already exist, with the associated values shown in the following table.

| STRING VALUE SUBKEY | VALUE                |
|---------------------|----------------------|
| UntrustedSites      | Disabled             |
| Internet            | AuthenticodeRequired |
| MyComputer          | AuthenticodeRequired |
| LocalIntranet       | AuthenticodeRequired |
| TrustedSites        | AuthenticodeRequired |

### To restrict the ClickOnce trust prompt programmatically

1. Create a Visual Basic or Visual C# console application in Visual Studio.
2. Open the *Program.vb* or *Program.cs* file for editing and add the following code.

```
Dim key As Microsoft.Win32.RegistryKey
key =
Microsoft.Win32.Registry.LocalMachine.CreateSubKey("SOFTWARE\MICROSOFT\.NETFramework\Security\TrustMa
nager\PromptingLevel")
key.SetValue("MyComputer", "AuthenticodeRequired")
key.SetValue("LocalIntranet", "AuthenticodeRequired")
key.SetValue("Internet", "AuthenticodeRequired")
key.SetValue("TrustedSites", "AuthenticodeRequired")
key.SetValue("UntrustedSites", "Disabled")
key.Close()
```

```
Microsoft.Win32.RegistryKey key;  
key =  
Microsoft.Win32.Registry.LocalMachine.CreateSubKey("SOFTWARE\\MICROSOFT\\.NETFramework\\Security\\TrustManager\\PromptingLevel");  
key.SetValue("MyComputer", "AuthenticodeRequired");  
key.SetValue("LocalIntranet", "AuthenticodeRequired");  
key.SetValue("Internet", "AuthenticodeRequired");  
key.SetValue("TrustedSites", "AuthenticodeRequired");  
key.SetValue("UntrustedSites", "Disabled");  
key.Close();
```

3. Build and run the application.

## Disable the ClickOnce trust prompt

You can disable the trust prompt so that end users are not given the option to install solutions that are not already trusted in their security policy.

**To disable the ClickOnce trust prompt by using the registry editor**

1. Open the registry editor:
  - a. Click **Start**, and then click **Run**.
  - b. In the **Open** box, type `regedit`, and then click **OK**.

2. Find the following registry key:

**\\HKEY\_LOCAL\_MACHINE\\SOFTWARE\\MICROSOFT\\.NETFramework\\Security\\TrustManager\\PromptingLevel**

If the key does not exist, create it.

3. Add the following subkeys as **String Value**, if they do not already exist, with the associated values shown in the following table.

| STRING VALUE SUBKEY | VALUE    |
|---------------------|----------|
| UntrustedSites      | Disabled |
| Internet            | Disabled |
| MyComputer          | Disabled |
| LocalIntranet       | Disabled |
| TrustedSites        | Disabled |

**To disable the ClickOnce trust prompt programmatically**

1. Create a Visual Basic or Visual C# console application in Visual Studio.
2. Open the *Program.vb* or *Program.cs* file for editing and add the following code.

```
Dim key As Microsoft.Win32.RegistryKey
key =
Microsoft.Win32.Registry.LocalMachine.CreateSubKey("SOFTWARE\MICROSOFT\ .NETFramework\Security\TrustMa
nager\PromptingLevel")
key.SetValue("MyComputer", "Disabled")
key.SetValue("LocalIntranet", "Disabled")
key.SetValue("Internet", "Disabled")
key.SetValue("TrustedSites", "Disabled")
key.SetValue("UntrustedSites", "Disabled")
key.Close()
```

```
Microsoft.Win32.RegistryKey key;
key =
Microsoft.Win32.Registry.LocalMachine.CreateSubKey("SOFTWARE\\MICROSOFT\\.NETFramework\\Security\\Tru
stManager\\PromptingLevel");
key.SetValue("MyComputer", "Disabled");
key.SetValue("LocalIntranet", "Disabled");
key.SetValue("Internet", "Disabled");
key.SetValue("TrustedSites", "Disabled");
key.SetValue("UntrustedSites", "Disabled");
key.Close();
```

3. Build and run the application.

## See also

- [Secure ClickOnce applications](#)
- [Code access security for ClickOnce applications](#)
- [ClickOnce and Authenticode](#)
- [Trusted application deployment overview](#)
- [How to: Enable ClickOnce security settings](#)
- [How to: Set a security zone for a ClickOnce application](#)
- [How to: Set custom permissions for a ClickOnce application](#)
- [How to: Debug a ClickOnce application with restricted permissions](#)
- [How to: Add a trusted publisher to a client computer for ClickOnce applications](#)
- [How to: Re-sign application and deployment manifests](#)

# How to: Sign setup files with SignTool.exe (ClickOnce)

3/5/2021 • 2 minutes to read • [Edit Online](#)

You can use *SignTool.exe* to sign a Setup program (*setup.exe*). This process helps ensure that tampered files are not installed on end-user computers.

By default, ClickOnce has signed manifests and a signed Setup program. However, if you want to change the parameters of the Setup program later, you must sign the Setup program later. If you change the parameters after the Setup program is signed, the signature becomes corrupted.

The following procedure generates unsigned manifests and an unsigned Setup program. Then, ClickOnce signing is enabled in Visual Studio to generate signed manifests. The Setup program is left unsigned so that the customer can sign the executable with their own certificate.

## To generate an unsigned Setup program and sign later

1. On the development computer, install the certificate that you want the sign the manifests with.
2. Select the project in **Solution Explorer**.
3. On the **Project** menu, click *ProjectName***Properties**.
4. In the **Signing** page, clear **Sign the ClickOnce manifests**.
5. In the **Publish** page, click **Prerequisites**.
6. Verify that all the prerequisites are selected, and then click **OK**.
7. In the **Publish** page, verify the publish settings and then click **Publish Now**.

The solution publishes the unsigned application manifest, unsigned deployment manifest, version-specific files, and unsigned Setup program to the publishing folder location.

8. In the **Publish** page, click **Prerequisites**.
9. In the **Prerequisites** dialog box, clear **Create setup program to install prerequisite components**.
10. In the **Publish** page, verify the publish settings and then click **Publish Now**.

The solution publishes the signed application manifest, signed deployment manifest, and version-specific files to the publishing folder location. The unsigned Setup program is not overwritten by the publish process.

11. At the customer site, open a command prompt.
12. Change to the directory that contains the *.exe* file.
13. Sign the *.exe* file with the following command:

```
signtool sign /sha1 CertificateHash Setup.exe  
signtool sign /f CertFileName Setup.exe
```

For example, to sign the Setup program, use one of the following commands:

```
signtool sign /sha1 CCB... Setup.exe  
signtool sign /f CertFileName Setup.exe
```

## See also

- [How to: Re-sign application and deployment manifests](#)

# Publish ClickOnce applications

3/5/2021 • 5 minutes to read • [Edit Online](#)

When publishing a ClickOnce application for the first time, publish properties can be set using the Publish Wizard. Only a few of the properties are available in the wizard; all other properties are set to their default values.

Subsequent changes to publish properties are made on the **Publish** page in the **Project Designer**.

## Publish Wizard

You can use the Publish Wizard to set the basic settings for publishing your application. This includes the following publishing properties:

- Publishing Folder Location - where Visual Studio will copy the files (local computer, network file share, FTP server, or Web site)
- Installation Folder Location - where end users will install from (network file share, FTP server, Web site, CD/DVD)
- Online or Offline availability - if end users can access the application with or without a network connection
- Update frequency - how often the application checks for new updates.

For more information, see [How to: Publish a ClickOnce application using the Publish Wizard](#).

## Publish Page

The **Publish** page of the **Project Designer** is used to configure properties for ClickOnce deployment. The following table lists topics.

| TITLE  | DESCRIPTION   |
|--|---|
| <a href="#">How to: Specify where Visual Studio copies the files</a>           | Describes how to set where Visual Studio puts the application files and manifests.  |
| <a href="#">How to: Specify the location where end users will install from</a> | Describes how to set the location where users go to download and install the application.   |
| <a href="#">How to: Specify the ClickOnce offline or online install mode</a>   | Describes how to set whether the application will be available offline or online.   |
| <a href="#">How to: Set the ClickOnce publish version</a>                      | Describes how to set the ClickOnce <b>Publish Version</b> property, which determines whether or not the application that you are publishing will be treated as an update. |
| <a href="#">How to: Automatically increment the ClickOnce publish version</a>  | Describes how to automatically increment the Revision number of the <b>PublishVersion</b> each time you publish the application.  |

For more information, see [Publish Page, Project Designer](#)

### Application Files dialog box



This dialog box allows you to specify how the files in your project are categorized for publishing, dynamic downloading, and updating. It contains a grid that lists the project files that are not excluded by default, or that have a download group.

To exclude files, mark files as data files or prerequisites, and create groups of files for conditional installation in the Visual Studio UI, see [How to: Specify which files are published by ClickOnce](#). You can also mark data files by using the Mage.exe. For more information, see [How to: Include a data file in a ClickOnce application](#).

### Prerequisites dialog box

This dialog box specifies which prerequisite components are installed, as well as how they are installed. For more information, see [How to: Install prerequisites with a ClickOnce application](#) and [Prerequisites dialog box](#).

### Application Updates dialog box

This dialog box specifies how the application installation should check for updates. For more information, see [How to: Manage updates for a ClickOnce application](#).

### Publish Options dialog box

The Publish Options dialog box specifies an application's deployment options.

| TITLE  | DESCRIPTION  |
|--|--|
| <a href="#">How to: Change the publish language for a ClickOnce application</a>                      | Describes how to specify a language and culture to match the localized version.  |
| <a href="#">How to: Specify a Start menu name for a ClickOnce application</a>                        | Describes how to change the display name for a ClickOnce application.  |
| <a href="#">How to: Specify a link for Technical Support</a>   | Describes how to set the <b>Support URL</b> property, which identifies a Web page or file share where users can go to get information about the application. |
| <a href="#">How to: Specify a Support URL for individual prerequisites in a ClickOnce deployment</a> | Demonstrated how to manually alter an application manifest to include individual support URLs for each prerequisite.   |
| <a href="#">How to: Specify a publish page for a ClickOnce application</a>                           | Describes how to generate and publish a default Web page (publish.htm) along with the application  |
| <a href="#">How to: Customize the ClickOnce default Web page</a>                                     | Describes how to customize the Web page that is automatically generated and published along with the application.  |
| <a href="#">How to: Enable AutoStart for CD installations</a>  | Describes how to enable AutoStart so that the ClickOnce application is automatically launched when the media is inserted.                                    |

## Related topics

| TITLE  | DESCRIPTION  |
|--|--|
| <a href="#">How to: Create file associations For a ClickOnce application</a>                 | Describes how to add file name extension support to a ClickOnce application.                   |
| <a href="#">How to: Retrieve query string information in an online ClickOnce application</a> | Demonstrates how to retrieve parameters passed in the URL used to run a ClickOnce application. |

| TITLE   | DESCRIPTION  |
|---|--|
| <a href="#">How to: Disable URL activation of ClickOnce applications by using the designer</a>  | Describes how to force users to start the application from the <b>Start</b> menu by using the designer.  |
| <a href="#">How to: Disable URL activation of ClickOnce applications</a>  | Describes how to force users to start the application from the <b>Start</b> menu.  |
| <a href="#">Walkthrough: Downloading assemblies on demand with the ClickOnce deployment API using the Designer</a>                            | Explains how to download application assemblies only when they are first used by the application using the designer.                                 |
| <a href="#">Walkthrough: Download assemblies on demand with the ClickOnce deployment API</a>  | Explains how to download application assemblies only when they are first used by the application.  |
| <a href="#">Walkthrough: Download satellite assemblies on demand with the ClickOnce deployment API</a>  | Describes how to mark your satellite assemblies as optional, and download only the assembly a client machine needs for its current culture settings. |
| <a href="#">Walkthrough: Manually deploy a ClickOnce application</a>  | Explains how to use .NET Framework utilities to deploy your ClickOnce application.   |
| <a href="#">Walkthrough: Manually deploy a ClickOnce application that does not require re-signing and that preserves branding information</a> | Explains how to use .NET Framework utilities to deploy your ClickOnce application without re-signing the manifests.                                  |
| <a href="#">How to: Configure projects to target platforms</a>  | Explains how to publish for a 64-bit processor by changing the <b>Target CPU</b> or <b>Platform target</b> property in your project.                 |
| <a href="#">Walkthrough: Enable a ClickOnce application to run on multiple .NET Framework versions</a>  | Explains how to enable a ClickOnce application to install and run on multiple versions of the .NET Framework.  |
| <a href="#">Walkthrough: Create a custom installer for a ClickOnce application</a>  | Explains how to create a custom installer to install a ClickOnce application.  |
| <a href="#">How to: Publish a WPF application with visual styles enabled</a>  | Provides step-by-step instructions to resolve an error that appears when you attempt to publish a WPF application that has visual styles enabled.    |

## See also

- [ClickOnce security and deployment](#)
- [ClickOnce reference](#)

# How to: Publish a ClickOnce application using the Publish Wizard

3/5/2021 • 3 minutes to read • [Edit Online](#)

To make a ClickOnce application available to users, you must publish it to a file share or path, FTP server, or removable media. You can publish the application by using the Publish Wizard; additional properties related to publishing are available on the **Publish** page of the **Project Designer**. For more information, see [Publishing ClickOnce applications](#).

Before you run the Publish Wizard, you should set the publishing properties appropriately. For example, if you want to designate a key to sign your ClickOnce application, you can do so on the **Signing** page of the **Project Designer**. For more information, see [Secure ClickOnce applications](#).

## NOTE

When you install more than one version of an application by using ClickOnce, the installation moves earlier versions of the application into a folder named *Archive*, in the publish location that you specify. Archiving earlier versions in this manner keeps the installation directory clear of folders from the earlier version.

## NOTE

The dialog boxes and menu commands you see might differ from those described in Help, depending on your active settings or edition. To change your settings, click **Import and Export Settings** on the **Tools** menu. For more information, see [Reset settings](#).

## To publish to a file share or path

1. In **Solution Explorer**, select the application project.
2. On the **Build** menu, click **Publish *Projectname***.  
  
The Publish Wizard appears.
3. In the **Where do you want to publish the application?** page, enter a valid FTP server address or a valid file path using one of the formats shown, and then click **Next**.
4. In the **How will users install the application?** page, select the location where users will go to install the application:
  - If users will install from a Web site, click **From a Web site** and enter a URL that corresponds to the file path entered in the previous step. Click **Next**. (This option is typically used when you specify an FTP address as the publishing location. Direct download from FTP is not supported. Therefore, you have to enter a URL here.)
  - If users will install the application directly from the file share, click **From a UNC path or file share**, and then click **Next**. (This is for publishing locations of the form `c:\deploy\myapp` or `\\server\myapp`.)
  - If users will install from removable media, click **From a CD-ROM or DVD-ROM**, and then click **Next**.

5. On the **Will the application be available offline?** page, click the appropriate option:

- If you want to enable the application to be run when the user is disconnected from the network, click **Yes, this application will be available online or offline**. A shortcut on the **Start** menu will be created for the application.
- If you want to run the application directly from the publish location, click **No, this application is only available online**. A shortcut on the **Start** menu will not be created.

Click **Next** to continue.

6. Click **Finish** to publish the application.

Publishing status is displayed in the status notification area.

## To publish to a CD-ROM or DVD-ROM

1. In **Solution Explorer**, right-click the application project and click **Properties**.

The **Project Designer** appears.

2. Click the **Publish** tab to open the **Publish** page in the **Project Designer**, and click the **Publish Wizard** button.

The Publish Wizard appears.

3. In the **Where do you want to publish the application?** page, enter the file path or FTP location where the application will be published, for example *d:\deploy*. Then click **Next** to continue.

4. On the **How will users install the application?** page, click **From a CD-ROM or DVD-ROM**, and then click **Next**.

### NOTE

If you want the installation to run automatically when the CD-ROM is inserted into the drive, open the **Publish** page in the **Project Designer** and click the **Options** button, and then, in the **Publish Options** wizard, select **For CD installations, automatically start Setup when CD is inserted**.

5. If you distribute your application on CD-ROM, you might want to provide updates from a Web site. In the **Where will the application check for updates?** page, choose an update option:

- If the application will check for updates, click **The application will check for updates from the following location** and enter the location where updates will be posted. This can be a file location, Web site, or FTP server.
- If the application will not check for updates, click **The application will not check for updates**.

Click **Next** to continue.

6. Click **Finish** to publish the application.

Publishing status is displayed in the status notification area.

### NOTE

After publishing is complete, you will have to use a CD-Rewriter or DVD-Rewriter to copy the files from the location specified in step 3 to the CD-ROM or DVD-ROM media.

## See also

- [ClickOnce security and deployment](#)
- [Secure ClickOnce applications](#)
- [Deploying an Office solution by using ClickOnce](#)

# Create ClickOnce applications for others to deploy

3/5/2021 • 9 minutes to read • [Edit Online](#)

Not all developers who are creating ClickOnce deployments plan to deploy the applications themselves. Many of them just package their application by using ClickOnce and then hand the files off to a customer, such as a large corporation. The customer becomes the one responsible for hosting the application on its network. This topic discusses some of the problems inherent in such deployments in versions of the .NET Framework prior to version 3.5. It then describes a new solution provided by using the new "use manifest for trust" feature in the .NET Framework 3.5. Finally, it concludes with recommended strategies for creating ClickOnce deployments for customers who are still using older versions of the .NET Framework.

## Issues involved in creating deployments for customers

Several issues occur when you plan to supply a deployment to a customer. The first issue concerns code signing. In order to be deployed across a network, the deployment manifest and application manifest of a ClickOnce deployment must both be signed with a digital certificate. This raises the question of whether to use the developer's certificate or the customer's certificate when signing the manifests.

The question of which certificate to use is critical, as a ClickOnce application's identity is based on the digital signature of the deployment manifest. If the developer signs the deployment manifest, it could lead to conflicts if the customer is a large company, and more than one division of the company deploys a customized version of the application.

For example, say that Adventure Works has a finance department and a human resources department. Both departments license a ClickOnce application from Microsoft Corporation that generates reports from data stored in a SQL database. Microsoft supplies each department with a version of the application that is customized for their data. If the applications are signed with the same Authenticode certificate, a user who tries to use both applications would encounter an error, as ClickOnce would regard the second application as being identical to the first. In this case, the customer could experience unpredictable and unwanted side effects that include the loss of any data stored locally by the application.

An additional problem related to code signing is the `deploymentProvider` element in the deployment manifest, which tells ClickOnce where to look for application updates. This element must be added to the deployment manifest prior to signing it. If this element is added afterward, the deployment manifest must be re-signed.

### Require the customer to sign the deployment manifest

One solution to this problem of non-unique deployments is to have the developer sign the application manifest, and the customer sign the deployment manifest. While this approach works, it introduces other issues. Since an Authenticode certificate must remain a protected asset, the customer cannot just give the certificate to the developer to sign the deployment. While the customer can sign the deployment manifest themselves by using tools freely available with the .NET Framework SDK, this may require more technical knowledge than the customer is willing or able to provide. In such cases, the developer usually creates an application, Web site, or other mechanism through which the customer can submit their version of the application for signing.

### The impact of customer signing on ClickOnce application security

Even if the developer and the customer agree that the customer should sign the application manifest, this raises other issues that surround the application's identity, especially as it applies to trusted application deployment. (For more information about this feature, see [Trusted application deployment overview](#).) Say that Adventure Works wants to configure its client computers so that any application provided to them by Microsoft Corporation runs with full trust. If Adventure Works signs the deployment manifest, then ClickOnce will use

Adventure Work's security signature to determine the trust level of the application.

## Create customer deployments by using application manifest for trust

ClickOnce in the .NET Framework 3.5 contains a new feature that gives developers and customers a new solution to the scenario of how the manifests should be signed. The ClickOnce application manifest supports a new element named `<useManifestForTrust>` that enables a developer to signify that the digital signature of the application manifest is what should be used for making trust decisions. The developer uses ClickOnce packaging tools—such as *Mage.exe*, *MageUI.exe*, and Visual Studio—to include this element in the application manifest, as well as to embed both their Publisher name and the name of the application in the manifest.

When using `<useManifestForTrust>`, the deployment manifest does not have to be signed with an Authenticode certificate issued by a certification authority. Instead, it can be signed with what is known as a self-signed certificate. A self-signed certificate is generated by either the customer or the developer by using standard .NET Framework SDK tools, and then applied to the deployment manifest by using the standard ClickOnce deployment tools. For more information, see [MakeCert](#).

Using a self-signed certificate for the deployment manifest presents several advantages. By eliminating the need for the customer to obtain or create their own Authenticode certificate, `<useManifestForTrust>` simplifies deployment for the customer, while allowing the developer to maintain their own branding identity on the application. The result is a set of signed deployments that are more secure and have unique application identities. This eliminates the potential conflict that may occur from deploying the same application to multiple customers.

For step-by-step information about how to create a ClickOnce deployment with `<useManifestForTrust>` enabled, see [Walkthrough: Manually deploy a ClickOnce application that does not require re-signing and that preserves branding information](#).

### How application manifest for trust works at run time

To get a better understanding of how using the application manifest for trust works at run time, consider the following example. A ClickOnce application that targets the .NET Framework 3.5 is created by Microsoft. The application manifest uses the `<useManifestForTrust>` element and is signed by Microsoft. Adventure Works signs the deployment manifest by using a self-signed certificate. Adventure Works clients are configured to trust any application signed by Microsoft.

When a user clicks a link to the deployment manifest, ClickOnce installs the application on the user's computer. The certificate and deployment information identify the application uniquely to ClickOnce on the client computer. If the user tries to install the same application again from a different location, ClickOnce can use this identity to determine that the application already exists on the client.

Next, ClickOnce examines the Authenticode certificate that is used to sign the application manifest, which determines the level of trust that ClickOnce will grant. Since Adventure Works has configured its clients to trust any application signed by Microsoft, this ClickOnce application is granted full trust. For more information, see [Trusted application deployment overview](#).

## Create customer deployments for earlier versions

What if a developer is deploying ClickOnce applications to customers who are using older versions of the .NET Framework? The following sections summarize several recommended solutions, together with the benefits and drawbacks of each.

### Sign deployments on behalf of customer

One possible deployment strategy is for the developer to create a mechanism to sign deployments on behalf of their customers, by using the customer's own private key. This prevents the developer from having to manage private keys or multiple deployment packages. The developer just provides the same deployment to each

customer. It is up to the customer to customize it for their environment by using the signing service.

One drawback to this method is the time and expense that are required to implement it. While such a service can be built by using the tools provided in the .NET Framework SDK, it will add more development time to the product life cycle.

As noted earlier in this topic, another drawback is that each customer's version of the application will have the same application identity, which could lead to conflicts. If this is a concern, the developer can change the Name field that is used when generating the deployment manifest to give each application a unique name. This will create a separate identity for each version of the application, and eliminate any potential identity conflicts. This field corresponds to the `-Name` argument for Mage.exe, and to the **Name** field on the **Name** tab in MageUI.exe.

For example, say that the developer has created an application named Application1. Instead of creating a single deployment with the Name field set to Application1, the developer can create several deployments with a customer-specific variation on this name, such as Application1-CustomerA, Application1-CustomerB, and so on.

### Deploy using a setup package

A second possible deployment strategy is to generate a Microsoft Setup project to perform the initial deployment of the ClickOnce application. This can be provided in one of several different formats: as an MSI deployment, as a setup executable (.EXE), or as a cabinet (.cab) file together with a batch script.

Using this technique, the developer would provide the customer a deployment that includes the application files, the application manifest, and a deployment manifest that serves as a template. The customer would run the setup program, which would prompt them for a deployment URL (the server or file share location from which users will install the ClickOnce application), as well as a digital certificate. The setup application may also choose to prompt for additional ClickOnce configuration options, such as update check interval. Once this information is gathered, the setup program would generate the real deployment manifest, sign it, and publish the ClickOnce application to the designated server location.

There are three ways that the customer can sign the deployment manifest in this situation:

1. The customer can use a valid certificate issued by a certification authority (CA).
2. As a variation on this approach, the customer can choose to sign their deployment manifest with a self-signed certificate. The drawback to this is that it will cause the application to display the words "Unknown Publisher" when the user is asked whether to install it. However, the benefit is that it prevents smaller customers from having to spend the time and money required for a certificate issued by a certification authority.
3. Finally, the developer can include their own self-signed certificate in the setup package. This introduces the potential problems with application identity discussed earlier in this topic.

The drawback to the setup deployment project method is the time and expense required to build a custom deployment application.

### Have customer generate deployment manifest

A third possible deployment strategy is to hand off only the application files and application manifest to the customer. In this scenario, the customer is responsible for using the .NET Framework SDK to generate and sign the deployment manifest.

The drawback of this method is that it requires the customer to install the .NET Framework SDK tools, and to have a developer or system administrator who is skilled at using them. Some customers may demand a solution that requires little or no technical effort on their part.

## See also

- [Deploy ClickOnce applications for testing and production servers without resigning](#)



- [Walkthrough: Manually deploying a ClickOnce application](#)
- [Walkthrough: Manually deploying a ClickOnce application that does not require re-signing and that preserves branding information](#)

# Deploy ClickOnce applications for testing and production servers without resigning

3/5/2021 • 3 minutes to read • [Edit Online](#)

This article describes a feature of ClickOnce introduced in the .NET Framework version 3.5 that enables the deployment of ClickOnce applications from multiple network locations without re-signing or changing the ClickOnce manifests.

## NOTE

Resigning is still the preferred method for deploying new versions of applications. Whenever possible, use the resigning method. For more information, see [Mage.exe \(Manifest Generation and Editing Tool\)](#).

Third-party developers and ISVs can opt in to this feature, making it easier for their customers to update their applications. This feature can be used in the following situations:

- When updating an application, not for the first installation of an application.
- When there is only one configuration of the application on a computer. For example, if an application is configured to point to two different databases, you cannot use this feature.

## Exclude deploymentProvider from deployment manifests

In the .NET Framework 2.0 and the .NET Framework 3.0, any ClickOnce application that installs on the system for offline availability must list a `deploymentProvider` in its deployment manifest. The `deploymentProvider` is often referred to as the update location; it is the location where ClickOnce checks for application updates. This requirement, along with the need for application publishers to sign their deployments, made it difficult for a company to update a ClickOnce application from a vendor or other third party. It also makes it more difficult to deploy the same application from multiple locations on the same network.

With changes that were made to ClickOnce in the .NET Framework 3.5, it is possible for a third party to provide a ClickOnce application to another organization, which can then deploy the application on its own network.

In order to take advantage of this feature, developers of ClickOnce applications must exclude `deploymentProvider` from their deployment manifests. This requirement means that you must exclude the `-providerUrl` argument when you create deployment manifests with Mage.exe. Or, if you are generating deployment manifests with MageUI.exe, you must make sure that the **Launch Location** text box on the **Application Manifest** tab is left blank.

## deploymentProvider and application updates

Starting with the .NET Framework 3.5, you no longer have to specify a `deploymentProvider` in your deployment manifest in order to deploy a ClickOnce application for both online and offline usage. This change supports the scenario where you need to package and sign the deployment yourself, but allow other companies to deploy the application over their networks.

The important point to remember is that applications that exclude a `deploymentProvider` cannot change their install location during updates, until they ship an update that includes the `deploymentProvider` tag again.

Here are two examples to clarify this point. In the first example, you publish a ClickOnce application that has no

`deploymentProvider` tag, and you ask users to install it from `http://www.adatum.com/MyApplication/`. If you decide you want to publish the next update of the application from `http://subdomain.adatum.com/MyApplication/`, you have no way of signifying this in the deployment manifest that resides in `http://www.adatum.com/MyApplication/`. You can do one of two things:

- Tell your users to uninstall the previous version, and install the new version from the new location.
- Include an update on `http://www.adatum.com/MyApplication/` that includes a `deploymentProvider` pointing to `http://www.adatum.com/MyApplication/`. Then, release another update later with `deploymentProvider` pointing to `http://subdomain.adatum.com/MyApplication/`.

In the second example, you publish a ClickOnce application that specifies `deploymentProvider`, and you then decide to remove it. Once the new version without `deploymentProvider` is downloaded to clients, you cannot redirect the path used for updates until you release a version of your application that has `deploymentProvider` restored. As with the first example, `deploymentProvider` must initially point to the current update location, not your new location. In this case, if you attempt to insert a `deploymentProvider` that refers to `http://subdomain.adatum.com/MyApplication/`, then the next update fails.

## Create a deployment

For step by step guidance on creating deployments that can be deployed from different network locations, see [Walkthrough: Manually deploy a ClickOnce application that does not require re-signing and that preserves branding information](#).

## See also

- [Mage.exe](#) (Manifest Generation and Editing Tool)
- [MageUI.exe](#) (Manifest Generation and Editing Tool, Graphical Client)

# Access local and remote data in ClickOnce applications

4/2/2021 • 7 minutes to read • [Edit Online](#)

Most applications consume or produce data. ClickOnce gives you a variety of options for reading and writing data, both locally and remotely.

## Local Data

With ClickOnce, you can load and store data locally by using any one of the following methods:

- ClickOnce Data Directory
- Isolated Storage
- Other Local Files

### ClickOnce data directory

Every ClickOnce application installed on a local computer has a data directory, stored in the user's Documents and Settings folder. Any file included in a ClickOnce application and marked as a "data" file is copied to this directory when an application is installed. Data files can be of any file type, the most frequently used being text, XML, and database files such as Microsoft Access .mdb files.

The data directory is intended for application-managed data, which is data that the application explicitly stores and maintains. All static, nondependency files not marked as "data" in the application manifest will instead reside in the Application Directory. This directory is where the application's executable (.exe) files and assemblies reside.

#### NOTE

When a ClickOnce application is uninstalled, its Data Directory is also removed. Never use the Data Directory to store end-user-managed data, such as documents.

### Mark data files in a ClickOnce distribution

To put an existing file inside the Data Directory, you must mark the existing file as a data file in your ClickOnce application's application manifest file. For more information, see [How to: Include a data file in a ClickOnce application](#).

### Read from and write to the data directory

Reading from the Data Directory requires that your ClickOnce application request Read permission; similarly, writing to the directory requires Write permission. Your application will automatically have this permission if it is configured to run with Full Trust. For more information about elevating permissions for your application by using either Permission Elevation or Trusted Application Deployment, see [Secure ClickOnce applications](#).

#### NOTE

If your organization does not use Trusted Application Deployment and has turned off Permission Elevation, asserting permissions will fail.

After your application has these permissions, it can access the Data Directory by using method calls on classes

within the [System.IO](#). You can obtain the path of the Data Directory within a Windows Forms ClickOnce application by using the [DataDirectory](#) property defined on the [CurrentDeployment](#) property of [ApplicationDeployment](#). This is the most convenient and recommended way to access your data. The following code example demonstrates how to do this for a text file named *CSV.txt* that you have included in your deployment as a data file.

```
if (ApplicationDeployment.IsNetworkDeployed)
{
    try
    {
        using (StreamReader sr = new StreamReader(ApplicationDeployment.CurrentDeployment.DataDirectory +
@"\CSV.txt"))
        {
            MessageBox.Show(sr.ReadToEnd());
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show("Could not read file. Error message: " + ex.Message);
    }
}
```

```
If (ApplicationDeployment.IsNetworkDeployed) Then
    Dim SR As StreamReader = Nothing

    Try
        SR = New StreamReader(ApplicationDeployment.CurrentDeployment.DataDirectory & "\CSV.txt")
        MessageBox.Show(SR.ReadToEnd())
    Catch Ex As Exception
        MessageBox.Show("Could not read file.")
    Finally
        SR.Close()
    End Try
End If
```

For more information on marking files in your deployment as data files, see [How to: Include a Data File in a ClickOnce Application](#).

You can also obtain the data directory path using the relevant variables on the [Application](#) class, such as [LocalUserAppDataPath](#).

Manipulating other types of files might require additional permissions. For example, if you want to use an Access database (*.mdb*) file, your application must assert full trust in order to use the relevant [System.Data](#) classes.

#### Data directory and application versions

Each version of an application has its own Data Directory, which is isolated from other versions. ClickOnce creates this directory regardless of whether any data files are included in the deployment so that the application has a location to create new data files at run time. When a new version of an application is installed, ClickOnce will copy all the existing data files from the previous version's Data Directory into the new version's Data Directory—whether they were included in the original deployment or created by the application.

ClickOnce will replace the older version of the file with the newer version of the server if a data file has a different hash value in the old version of the application as in the new version. Also, if the earlier version of the application created a new file that has the same name as a file included in the new version's deployment, ClickOnce will overwrite the old version's file with the new file. In both cases, the old files will be included in a subdirectory inside the data directory named `.pre`, so that the application can still access the old data for migration purposes.

If you need finer-grained migration of data, you can use the ClickOnce Deployment API to perform custom migration from the old Data Directory to the new Data Directory. You will have to test for an available download by using [IsFirstRun](#), download the update using [Update](#) or [UpdateAsync](#), and do any custom data migration work in your own after the update is finished.

### Isolated storage

Isolated Storage provides an API for creating and accessing files by using a simple API. The actual location of the stored files is hidden from both the developer and the user.

Isolated Storage works in all versions of the .NET Framework. Isolated Storage also works in partially trusted applications without the need for additional permission grants. You should use Isolated Storage if your application must run in partial trust, but must maintain application-specific data.

For more information, see [Isolated Storage](#).

### Other local files

If your application must work with or save end-user data such as reports, images, music, and so on, your application will require [FileIOPermission](#) to read and write data to the local file system.

## Remote data

At some point, your application will likely have to retrieve information from a remote Web site, such as customer data or market information. This section discusses the most common techniques for retrieving remote data.

### Access files with HTTP

You can access data from a Web server by using either the [WebClient](#) or the [HttpWebRequest](#) class in the [System.Net](#) namespace. The data can be either static files or ASP.NET applications that return raw text or XML data. If your data is in XML format, the fastest way to retrieve the data is by using the [XmlDocument](#) class, whose [Load](#) method takes a URL as an argument. For an example, see [Read an XML document into the DOM](#).

You have to consider security when your application accesses remote data over HTTP. By default, your ClickOnce application's access to network resources may be restricted, depending on how your application was deployed. These restrictions are applied to prevent malicious programs from gaining access to privileged remote data or from using a user's computer to attack other computers on the network.

The following table lists the deployment strategies you might use and their default Web permissions.

| DEPLOYMENT TYPE    | DEFAULT NETWORK PERMISSIONS   |
|--------------------|---|
| Web Install        | Can only access the Web server from which the application was installed |
| File Share Install | Cannot access any Web servers   |
| CD-ROM Install     | Can access any Web servers  |

If your ClickOnce application cannot access a Web server because of security restrictions, the application must assert [WebPermission](#) for that Web site. For more information about increasing security permissions for a ClickOnce application, see [Secure ClickOnce applications](#).

### Access data through an XML Web service

If you expose your data as an XML Web service, you can access the data by using an XML Web service proxy. The proxy is a .NET Framework class you create by using either Visual Studio. The operations of the XML Web service—such as retrieving customers, placing orders, and so on—are exposed as methods on the proxy. This

makes Web services much easier to use than raw text or XML files.

If your XML Web service operates over HTTP, the service will be bound by the same security restrictions as the [WebClient](#) and [HttpWebRequest](#) classes.

### **Access a database directly**

You can use the classes within the [System.Data](#) namespace to establish direct connections with a database server such as SQL Server on your network, but you must account for the security issues. Unlike HTTP requests, database connection requests are always forbidden by default under partial trust; you will only have such permission by default if you install your ClickOnce application from a CD-ROM. This gives your application full trust. To enable access to a specific SQL Server database, your application must request [SqlClientPermission](#) to it; to enable access to a database other than SQL Server, it must request [OleDbPermission](#).

Most of the time, you will not have to access the database directly, but will access it instead through a Web server application written in ASP.NET or an XML Web service. Accessing the database in this manner is frequently the best method if your ClickOnce application is deployed from a Web server. You can access the server in partial trust without elevating your application's permissions.

## **See also**

- [How to: Include a data file in a ClickOnce application](#)

# Deploy COM components with ClickOnce

3/5/2021 • 9 minutes to read • [Edit Online](#)

Deployment of legacy COM components has traditionally been a difficult task. Components need to be globally registered and thus can cause undesirable side effects between overlapping applications. This situation is generally not a problem in .NET Framework applications because components are completely isolated to an application or are side-by-side compatible. Visual Studio allows you to deploy isolated COM components on the Windows XP or higher operating system.

ClickOnce provides an easy and safe mechanism for deploying your .NET applications. However, if your applications use legacy COM components, you will need to take additional steps for deploying them. This topic describes how to deploy isolated COM components and reference native components (for example, from Visual Basic 6.0 or Visual C++).

For more information on deploying isolated COM components, see [Simplify App Deployment with ClickOnce and Registration-Free COM](#).

## Registration-free COM

Registration-free COM is a new technology for deploying and activating isolated COM components. It works by putting all the component's type-library and registration information that is typically installed into the system registry into an XML file called a manifest, stored in the same folder as the application.

Isolating a COM component requires that it be registered on the developer's machine, but it does not have to be registered on the end user's computer. To isolate a COM component, all you need to do is set its reference's **Isolated** property to **True**. By default, this property is set to **False**, indicating that it should be treated as a registered COM reference. If this property is **True**, it causes a manifest to be generated for this component at build time. It also causes the corresponding files to be copied to the application folder during installation.

When the manifest generator encounters an isolated COM reference, it enumerates all of the `Coclass` entries in the component's type library, matching each entry with its corresponding registration data, and generating manifest definitions for all the COM classes in the type library file.

## Deploy registration-free COM components using ClickOnce

ClickOnce deployment technology is well-suited for deploying isolated COM components, because both ClickOnce and registration-free COM require that a component have a manifest in order to be deployed.

Typically, the author of the component should provide a manifest. If not, however, Visual Studio is capable of generating a manifest automatically for a COM component. The manifest generation is performed during the ClickOnce Publish process; for more information, see [Publishing ClickOnce Applications](#). This feature also allows you to leverage legacy components that you authored in earlier development environments such as Visual Basic 6.0.

There are two ways that ClickOnce deploys COM components:

- Use the bootstrapper to deploy your COM components; this works on all supported platforms.
- Use native component isolation (also known as registration-free COM) deployment. However, this will only work on a Windows XP or higher operating system.

### Example of isolating and deploying a simple COM component

In order to demonstrate registration-free COM component deployment, this example will create a Windows-



based application in Visual Basic that references an isolated native COM component created using Visual Basic 6.0, and deploy it using ClickOnce.

First you will need to create the native COM component:

To create a native COM component

1. Using Visual Basic 6.0, from the **File** menu, click **New**, then **Project**.
2. In the **New Project** dialog box, select the **Visual Basic** node and select an **ActiveX DLL** project. In the **Name** box, type `VB6Hello`.

#### NOTE

Only ActiveX DLL and ActiveX Control project types are supported with registration-free COM; ActiveX EXE and ActiveX Document project types are not supported.

3. In **Solution Explorer**, double-click **Class1.vb** to open the text editor.
4. In **Class1.vb**, add the following code after the generated code for the `New` method:

```
Public Sub SayHello()  
    MsgBox "Message from the VB6Hello COM component"  
End Sub
```

5. Build the component. From the **Build** menu, click **Build Solution**.

#### NOTE

Registration-free COM supports only DLLs and COM controls project types. You cannot use EXEs with registration-free COM.

Now you can create a Windows-based application and add a reference to the COM component to it.

To create a Windows-based application using a COM component

1. Using Visual Basic, from the **File** menu, click **New**, then **Project**.
2. In the **New Project** dialog box, select the **Visual Basic** node and select **Windows Application**. In the **Name** box, type `RegFreeComDemo`.
3. In **Solution Explorer**, click the **Show All Files** button to display the project references.
4. Right-click the **References** node and select **Add Reference** from the context menu.
5. In the **Add Reference** dialog box, click the **Browse** tab, navigate to `VB6Hello.dll`, then select it.  
  
A **VB6Hello** reference appears in the references list.
6. Point to the **Toolbox**, select a **Button** control, and drag it to the **Form1** form.
7. In the **Properties** window, set the button's **Text** property to **Hello**.
8. Double-click the button to add handler code, and in the code file, add code so that the handler reads as follows:

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles  
    Button1.Click  
        Dim VbObj As New VB6Hello.Class1  
        VbObj.SayHello()  
    End Sub
```

9. Run the application. From the **Debug** menu, click **Start Debugging**.

Next you need to isolate the control. Each COM component that your application uses is represented in your project as a COM reference. These references are visible under the **References** node in the **Solution Explorer** window. (Notice that you can add references either directly using the **Add Reference** command on the **Project** menu, or indirectly by dragging an ActiveX control onto your form.)

The following steps show how to isolate the COM component and publish the updated application containing the isolated control:

**To isolate a COM component**

1. In **Solution Explorer**, in the **References** node, select the **VB6Hello** reference.
2. In the **Properties** window, change the value of the **Isolated** property from **False** to **True**.
3. From the **Build** menu, click **Build Solution**.

Now, when you press F5, the application works as expected, but it is now running under registration-free COM. In order to prove this, try unregistering the VB6Hello.dll component and running RegFreeComDemo1.exe outside of the Visual Studio IDE. This time when the button is clicked, it still works. If you temporarily rename the application manifest, it will again fail.

#### NOTE

You can simulate the absence of a COM component by temporarily unregistering it. Open a command prompt, go to your system folder by typing `cd /d %windir%\system32`, then unregister the component by typing `regsvr32 /u VB6Hello.dll`. You can register it again by typing `regsvr32 VB6Hello.dll`.

The final step is to publish the application using ClickOnce:

**To publish an application update with an isolated COM component**

1. From the **Build** menu, click **Publish RegFreeComDemo**.

The Publish Wizard appears.

2. In the Publish Wizard, specify a location on the local computer's disk where you can access and examine the published files.
3. Click **Finish** to publish the application.

If you examine the published files, you will note that the sysmon.ocx file is included. The control is totally isolated to this application, meaning that if the end user's machine has another application using a different version of the control, it cannot interfere with this application.

## Reference native assemblies

Visual Studio supports references to native Visual Basic 6.0 or C++ assemblies; such references are called native references. You can tell whether a reference is native by verifying that its **File Type** property is set to **Native** or **ActiveX**.

To add a native reference, use the **Add Reference** command, then browse to the manifest. Some components place the manifest inside the DLL. In this case, you can simply choose the DLL itself and Visual Studio will add it

as a native reference if it detects that the component contains an embedded manifest. Visual Studio will also automatically include any dependent files or assemblies listed in the manifest if they are in the same folder as the referenced component.

COM control isolation makes it easy to deploy COM components that do not already have manifests. However, if a component is supplied with a manifest, you can reference the manifest directly. In fact, you should always use the manifest supplied by the author of the component wherever possible rather than using the **Isolated** property.

## Limitations of registration-free COM component deployment

Registration-free COM provides clear advantages over traditional deployment techniques. However, there are a few limitations and caveats that should also be pointed out. The greatest limitation is that it only works on Windows XP or higher. The implementation of registration-free COM required changes to the way in which components are loaded in the core operating system. Unfortunately, there is no down-level support layer for registration-free COM.

Not every component is a suitable candidate for registration-free COM. A component is not a suitable if any of the following are true:

- The component is an out-of-process server. EXE servers are not supported; only DLLs are supported.
- The component is part of the operating system, or is a system component, such as XML, Internet Explorer, or Microsoft Data Access Components (MDAC). You should follow the redistribution policy of the component author; check with your vendor.
- The component is part of an application, such as Microsoft Office. For example, you should not attempt to isolate Microsoft Excel Object Model. This is part of Office and can only be used on a computer with the full Office product installed.
- The component is intended for use as an add-in or a snap-in, for example an Office add-in or a control in a Web browser. Such components typically require some kind of registration scheme defined by the hosting environment that is beyond the scope of the manifest itself.
- The component manages a physical or virtual device for the system, for example, a device driver for a print spooler.
- The component is a Data Access redistributable. Data applications generally require a separate Data Access redistributable to be installed before they can run. You should not attempt to isolate components such as the Microsoft ADO Data Control, Microsoft OLE DB, or Microsoft Data Access Components (MDAC). Instead, if your application uses MDAC or SQL Server Express, you should set them as prerequisites; see [How to: Install Prerequisites with a ClickOnce Application](#).

In some cases, it may be possible for the developer of the component to redesign it for registration-free COM. If this is not possible, you can still build and publish applications that depend on them through the standard registration scheme using the Bootstrapper. For more information, see [Creating Bootstrapper Packages](#).

A COM component can only be isolated once per application. For example, you can't isolate the same COM component from two different **Class Library** projects that are part of the same application. Doing so will result in a build warning, and the application will fail to load at run time. In order to avoid this problem, Microsoft recommends that you encapsulate COM components in a single class library.

There are several scenarios in which COM registration is required on the developer's machine, even though the application's deployment does not require registration. The `Isolated` property requires that the COM component be registered on the developer's machine in order to auto-generate the manifest during the build. There are no registration-capturing capabilities that invoke the self-registration during

the build. Also, any classes not explicitly defined in the type library will not be reflected in the manifest. When using a COM component with a pre-existing manifest, such as a native reference, the component may not need to be registered at development time. However, registration is required if the component is an ActiveX control and you want to include it in the **Toolbox** and the Windows Forms designer.

## See also

- [ClickOnce security and deployment](#)

# Build ClickOnce applications from the command line

3/5/2021 • 8 minutes to read • [Edit Online](#)

In Visual Studio, you can build projects from the command line, even if they are created in the integrated development environment (IDE). In fact, you can rebuild a project created with Visual Studio on another computer that has only the .NET Framework installed. This allows you to reproduce a build using an automated process, for example, in a central build lab or using advanced scripting techniques beyond the scope of building the project itself.

## Use MSBuild to reproduce .NET Framework ClickOnce application deployments

When you invoke `msbuild /target:publish` at the command line, it tells the MSBuild system to build the project and create a ClickOnce application in the publish folder. This is equivalent to selecting the **Publish** command in the IDE.

This command executes *msbuild.exe*, which is on the path in the Visual Studio command-prompt environment.

A "target" is an indicator to MSBuild on how to process the command. The key targets are the "build" target and the "publish" target. The build target is the equivalent to selecting the Build command (or pressing F5) in the IDE. If you only want to build your project, you can achieve that by typing `msbuild`. This command works because the build target is the default target for all projects generated by Visual Studio. This means you do not explicitly need to specify the build target. Therefore, typing `msbuild` is the same operation as typing

```
msbuild /target:build .
```

The `/target:publish` command tells MSBuild to invoke the publish target. The publish target depends on the build target. This means that the publish operation is a superset of the build operation. For example, if you made a change to one of your Visual Basic or C# source files, the corresponding assembly would automatically be rebuilt by the publish operation.

For information on generating a full ClickOnce deployment using the Mage.exe command-line tool to create your ClickOnce manifest, see [Walkthrough: Manually deploy a ClickOnce application](#).

## Create and build a basic ClickOnce application with MSBuild

### To create and publish a ClickOnce project

1. Open Visual Studio and create a new project.

Choose the **Windows Desktop Application** project template and name the project `CmdLineDemo`.

2. From the **Build** menu, click the **Publish** command.

This step ensures that the project is properly configured to produce a ClickOnce application deployment.

The Publish Wizard appears.

3. In the Publish Wizard, click **Finish**.

Visual Studio generates and displays the default Web page, called *Publish.htm*.

4. Save your project, and make note of the folder location in which it is stored.

The steps above create a ClickOnce project which has been published for the first time. Now you can

reproduce the build outside of the IDE.

**To reproduce the build from the command line**

1. Exit Visual Studio.
2. From the Windows **Start** menu, click **All Programs**, then **Microsoft Visual Studio**, then **Visual Studio Tools**, then **Visual Studio Command Prompt**. This should open a command prompt in the root folder of the current user.
3. In the **Visual Studio Command Prompt**, change the current directory to the location of the project you just built above. For example, type `chdir My Documents\Visual Studio\Projects\CmdLineDemo`.
4. To remove the existing files produced in "To create and publish a ClickOnce project," type `rmdir /s publish`.

This step is optional, but it ensures that the new files were all produced by the command-line build.

5. Type `msbuild /target:publish`.

The above steps will produce a full ClickOnce application deployment in a subfolder of your project named **Publish**. *CmdLineDemo.application* is the ClickOnce deployment manifest. The folder *CmdLineDemo\_1.0.0.0* contains the files *CmdLineDemo.exe* and *CmdLineDemo.exe.manifest*, the ClickOnce application manifest. *Setup.exe* is the bootstrapper, which by default is configured to install the .NET Framework. The DotNetFX folder contains the redistributables for the .NET Framework. This is the entire set of files you need to deploy your application over the Web or via UNC or CD/DVD.

**NOTE**

The MSBuild system uses the **PublishDir** option to specify the location for output, for example

```
msbuild /t:publish /p:PublishDir="<specific location>"
```

## Build .NET ClickOnce applications from the command line

Building .NET ClickOnce applications from the command line is a similar experience except, you need to provide an additional property for the publish profile on the MSBuild command line. The easiest way to create a publish profile is by using Visual Studio. See [Deploy a .NET Windows application using ClickOnce](#) for more information.

Once you have the publish profile created, you can provide the pubxml file as a property on the msbuild command line. For example:

```
msbuild /t:publish /p:PublishProfile=<pubxml file> /p:PublishDir="<specific location>"
```

## Publish properties

When you publish the application in the above procedures, the following properties are inserted into your project file by the Publish Wizard or in the publish profile file for .NET Core 3.1, or later projects. These properties directly influence how the ClickOnce application is produced.

In *CmdLineDemo.vbproj* / *CmdLineDemo.csproj*:

```
<AssemblyOriginatorKeyFile>WindowsApplication3.snk</AssemblyOriginatorKeyFile>
<GenerateManifests>true</GenerateManifests>
<TargetZone>LocalIntranet</TargetZone>
<PublisherName>Microsoft</PublisherName>
<ProductName>CmdLineDemo</ProductName>
<PublishUrl>http://localhost/CmdLineDemo</PublishUrl>
<Install>true</Install>
<ApplicationVersion>1.0.0.*</ApplicationVersion>
<ApplicationRevision>1</ApplicationRevision>
<UpdateEnabled>true</UpdateEnabled>
<UpdateRequired>false</UpdateRequired>
<UpdateMode>Foreground</UpdateMode>
<UpdateInterval>7</UpdateInterval>
<UpdateIntervalUnits>Days</UpdateIntervalUnits>
<UpdateUrlEnabled>false</UpdateUrlEnabled>
<IsWebBootstrapper>true</IsWebBootstrapper>
<BootstrapperEnabled>true</BootstrapperEnabled>
```

For .NET Framework projects, you can override any of these properties at the command line without altering the project file itself. For example, the following will build the ClickOnce application deployment without the bootstrapper:

```
msbuild /target:publish /property:BootstrapperEnabled=false
```

For .NET Core 3.1, or later, projects these settings are provided in the pubxml file.

Publishing properties are controlled in Visual Studio from the **Publish**, **Security**, and **Signing** property pages of the **Project Designer**. Below is a description of the publishing properties, along with an indication of how each is set in the various property pages of the application designer:

#### NOTE

For .NET Windows desktop projects, these settings are now found in the Publish Wizard

- **AssemblyOriginatorKeyFile** determines the key file used to sign your ClickOnce application manifests. This same key may also be used to assign a strong name to your assemblies. This property is set on the **Signing** page of the **Project Designer**.

For .NET windows applications, this setting remains in the project file

The following properties are set on the **Security** page:

- **Enable ClickOnce Security Settings** determines whether ClickOnce manifests are generated. When a project is initially created, ClickOnce manifest generation is off by default. The wizard will automatically turn this flag on when you publish for the first time.
- **TargetZone** determines the level of trust to be emitted into your ClickOnce application manifest. Possible values are "Internet", "LocalIntranet", and "Custom". Internet and LocalIntranet will cause a default permission set to be emitted into your ClickOnce application manifest. LocalIntranet is the default, and it basically means full trust. Custom specifies that only the permissions explicitly specified in the base *app.manifest* file are to be emitted into the ClickOnce application manifest. The *app.manifest* file is a partial manifest file that contains just the trust information definitions. It is a hidden file, automatically added to your project when you configure permissions on the **Security** page.

-

## NOTE

For .NET Core 3.1, or later, Windows desktop projects, these Security settings are not supported.

The following properties are set on the **Publish** page:

- `PublishUrl` is the location where the application will be published to in the IDE. It is inserted into the ClickOnce application manifest if neither the `InstallUrl` or `UpdateUrl` property is specified.
- `ApplicationVersion` specifies the version of the ClickOnce application. This is a four-digit version number. If the last digit is a "\*", then the `ApplicationRevision` is substituted for the value inserted into the manifest at build time.
- `ApplicationRevision` specifies the revision. This is an integer which increments each time you publish in the IDE. Notice that it is not automatically incremented for builds performed at the command-line.
- `Install` determines whether the application is an installed application or a run-from-Web application.
- `InstallUrl` (not shown) is the location where users will install the application from. If specified, this value is burned into the *setup.exe* bootstrapper if the `IsWebBootstrapper` property is enabled. It is also inserted into the application manifest if the `UpdateUrl` is not specified.
- `SupportUrl` (not shown) is the location linked in the **Add/Remove Programs** dialog box for an installed application.

The following properties are set in the **Application Updates** dialog box, accessed from the **Publish** page.

- `UpdateEnabled` indicates whether the application should check for updates.
- `UpdateMode` specifies either Foreground updates or Background updates.

For .NET Core 3.1, or later, projects, Background is not supported.

- `UpdateInterval` specifies how frequently the application should check for updates.

For .NET Core 3.1, or later, this setting is not supported.

- `UpdateIntervalUnits` specifies whether the `UpdateInterval` value is in units of hours, days, or weeks.

For .NET Core 3.1, or later, this setting is not supported.

- `UpdateUrl` (not shown) is the location from which the application will receive updates. If specified, this value is inserted into the application manifest.

The following properties are set in the **Publish Options** dialog box, accessed from the **Publish** page.

- `PublisherName` specifies the name of the publisher displayed in the prompt shown when installing or running the application. In the case of an installed application, it is also used to specify the folder name on the **Start** menu.
- `ProductName` specifies the name of the product displayed in the prompt shown when installing or running the application. In the case of an installed application, it is also used to specify the shortcut name on the **Start** menu.

The following properties are set in the **Prerequisites** dialog box, accessed from the **Publish** page.

- `BootstrapperEnabled` determines whether to generate the *setup.exe* bootstrapper.



- `IsWebBootstrapper` determines whether the *setup.exe* bootstrapper works over the Web or in disk-based mode.

## InstallURL, SupportUrl, PublishURL, and UpdateURL

The following table shows the four URL options for ClickOnce deployment.

| URL OPTION              | DESCRIPTION  |
|-------------------------|--|
| <code>PublishURL</code> | Required if you are publishing your ClickOnce application to a Web site.   |
| <code>InstallURL</code> | Optional. Set this URL option if the installation site is different than the <code>PublishURL</code> . For example, you could set the <code>PublishURL</code> to an FTP path and set the <code>InstallURL</code> to a Web URL. |
| <code>SupportURL</code> | Optional. Set this URL option if the support site is different than the <code>PublishURL</code> . For example, you could set the <code>SupportURL</code> to your company's customer support Web site.                          |
| <code>UpdateURL</code>  | Optional. Set this URL option if the update location is different than the <code>InstallURL</code> . For example, you could set the <code>PublishURL</code> to an FTP path and set the <code>UpdateURL</code> to a Web URL.    |

## See also

- [GenerateBootstrapper](#)
- [GenerateApplicationManifest](#)
- [GenerateDeploymentManifest](#)
- [ClickOnce security and deployment](#)
- [Walkthrough: Manually deploy a ClickOnce application](#)

# How to: Specify where Visual Studio copies the files

3/5/2021 • 2 minutes to read • [Edit Online](#)

When you publish an application by using ClickOnce, the `Publish Location` property specifies the location where the application files and manifest are put. This can be a file path or the path to an FTP server.

You can specify the `Publish Location` property on the **Publish** page of the **Project Designer**, or by using the Publish Wizard. For more information, see [How to: Publish a ClickOnce Application using the Publish Wizard](#).

## NOTE

When you install more than one version of an application by using ClickOnce, the installation moves earlier versions of the application into a folder named Archive, in the publish location that you specify. Archiving earlier versions in this manner keeps the installation directory clear of folders from the earlier version.

## To specify a publishing location

1. With a project selected in **Solution Explorer**, on the **Project** menu, click **Properties**.
2. Click the **Publish** tab.
3. In the **Publish Location** field, enter the publishing location by using one of the following formats:
  - To publish to a file share or disk path, enter the path by using either a UNC path (`\\Server\ApplicationName`) or a file path (`C:\Deploy\ApplicationName`).
  - To publish to an FTP server, enter the path using the format `ftp://ftp.microsoft.com/<ApplicationName>`.

Note that text must be present in the **Publishing Location** box in order for the Browse (...) button to work.

## See also

- [Publishing ClickOnce applications](#)
- [How to: Publish a ClickOnce application using the Publish Wizard](#)

# How to: Specify the location where end users will install from

3/5/2021 • 2 minutes to read • [Edit Online](#)

When publishing a ClickOnce application, the location where users go to download and install the application is not necessarily the location where you initially publish the application. For example, in some organizations a developer might publish an application to a staging server, and then an administrator would move the application to a Web server.

In this case, you can use the `Installation URL` property to specify the Web server where users will go to download the application. This is necessary so that the application manifest knows where to look for updates.

The `Installation URL` property can be set on the **Publish** page of the **Project Designer**.

## NOTE

The `Installation URL` property can also be set using the **PublishWizard**. For more information, see [How to: Publish a ClickOnce application using the Publish Wizard](#).

## To specify an Installation URL

1. With a project selected in **Solution Explorer**, on the **Project** menu, click **Properties**.
2. Click the **Publish** tab.
3. In the Installation URL field, enter the installation location using a fully qualified URL using the format `https://www.contoso.com/ApplicationName`, or a UNC path using the format `\\Server\ApplicationName`.

## See also

- [How to: Specify where Visual Studio copies the files](#)
- [Publishing ClickOnce applications](#)
- [How to: Publish a ClickOnce application using the Publish Wizard](#)

# How to: Specify the ClickOnce offline or online install mode

3/5/2021 • 2 minutes to read • [Edit Online](#)

The `Install Mode` for a ClickOnce application determines whether the application will be available offline or online. When you choose **The application is available online only**, the user must have access to the ClickOnce publishing location (either a Web page or a file share) in order to run the application. When you choose **The application is available offline as well**, the application adds entries to the **Start** menu and the **Add or Remove Programs** dialog box; the user is able to run the application when they are not connected.

The `Install Mode` can be set on the **Publish** page of the **Project Designer**.

## NOTE

The `Install Mode` can also be set using the Publish wizard. For more information, see [How to: Publish a ClickOnce application using the Publish Wizard](#).

### To make a ClickOnce application available online only

1. With a project selected in **Solution Explorer**, on the **Project** menu, click **Properties**.
2. Click the **Publish** tab.
3. In the **Install Mode and Settings** area, click the **The application is available online only** option button.

### To make a ClickOnce application available online or offline

1. With a project selected in **Solution Explorer**, on the **Project** menu, click **Properties**.
2. Click the **Publish** tab.
3. In the **Install Mode and Settings** area, click the **The application is available offline as well** option button.

When installed, the application adds entries to the **Start** menu and to **Add or Remove Programs** in Control Panel.

## See also

- [Publish ClickOnce applications](#)
- [How to: Publish a ClickOnce application using the Publish Wizard](#)
- [Choose a ClickOnce deployment strategy](#)

# How to: Set the ClickOnce publish version

3/5/2021 • 2 minutes to read • [Edit Online](#)

The ClickOnce `Publish Version` property determines whether or not the application that you are publishing will be treated as an update. Each time version is incremented, the application will be published as an update.

The `Publish Version` property can be set on the **Publish** page of the **Project Designer**.

## NOTE

There is a project option that will automatically increment the `Publish Version` property each time the application is published; this option is enabled by default. For more information, see [How to: Automatically Increment the ClickOnce Publish Version](#).

## To change the publish version

1. With a project selected in **Solution Explorer**, on the **Project** menu click **Properties**.
2. Click the **Publish** tab.
3. In **Publish Version** field, increment the **Major**, **Minor**, **Build**, or **Revision** version numbers.

## NOTE

You should never decrement a version number; doing so could cause unpredictable update behavior.

## See also

- [Choose a ClickOnce update strategy](#)
- [How to: Automatically increment the ClickOnce publish version](#)
- [Publish ClickOnce applications](#)
- [How to: Publish a ClickOnce application using the Publish Wizard](#)

# How to: Automatically increment the ClickOnce publish version

3/5/2021 • 2 minutes to read • [Edit Online](#)

When publishing a ClickOnce application, changing the `Publish Version` property causes the application to be published as an update. By default, Visual Studio automatically increments the `Revision` number of the `Publish Version` each time you publish the application.

You can disable this behavior on the **Publish** page of the **Project Designer**.

## NOTE

The dialog boxes and menu commands you see might differ from those described in Help depending on your active settings or edition. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see [Reset settings](#).

## To disable automatically incrementing the publish version

1. With a project selected in **Solution Explorer**, on the **Project** menu, click **Properties**.
2. Click the **Publish** tab.
3. In the **Publish Version** section, clear the **Automatically increment revision with each release** check box.

## See also

- [How to: Set the ClickOnce publish version](#)
- [Publish ClickOnce applications](#)
- [How to: Publish a ClickOnce application using the Publish Wizard](#)

# How to: Specify which files are published by ClickOnce

3/5/2021 • 3 minutes to read • [Edit Online](#)

When publishing a ClickOnce application, all non-code files in the project are deployed along with the application. In some cases, you may not want or need to publish certain files, or you may want to install certain files based on conditions. Visual Studio provides the capabilities to exclude files, mark files as data files or prerequisites, and create groups of files for conditional installation.

Files for a ClickOnce application are managed in the **Application Files** dialog box, accessible from the **Publish** page of the **Project Designer**.

Initially, there is a single file group named **(Required)**. You can create additional file groups and assign files to them. You cannot change the **Download Group** for files that are required for the application to run. For example, the application's .exe or files marked as data files must belong to the **(Required)** group.

The default publish status value of a file is tagged with **(Auto)**. For example, the application's .exe has a publish status of **Include (Auto)** by default.

Files with the **Build Action** property set to **Content** are designated as application files and will be marked as included by default. They can be included, excluded, or marked as data files. The exceptions are as follows:

- Data files such as SQL Database (.mdf and .mdb) files and XML files will be marked as data files by default.
- References to assemblies (.dll files) are designated as follows when you add the reference: If **Copy Local** is **False**, it is marked by default as a prerequisite assembly (**Prerequisite (Auto)**) that must be present in the GAC before the application is installed. If **Copy Local** is **True**, the assembly is marked by default as an application assembly (**Include (Auto)**) and will be copied into the application folder at installation. A COM reference will appear in the **Application Files** dialog box (as an .ocx file) only if its **Isolated** property is set to **True**. By default, it will be included.

## To add files to the Application Files dialog box

1. Select a data file in **Solution Explorer**.
2. In the **Properties** window, change the **Build Action** property to the **Content** value.

## To exclude files from ClickOnce publishing

1. With a project selected in **Solution Explorer**, on the **Project** menu, click **Properties**.
2. Click the **Publish** tab.
3. Click the **Application Files** button to open the **Application Files** dialog box.
4. In the **Application Files** dialog box, select the file that you wish to exclude.
5. In the **Publish Status** field, select **Exclude** from the drop-down list.

## To mark files as data files

1. With a project selected in **Solution Explorer**, on the **Project** menu, click **Properties**.
2. Click the **Publish** tab.
3. Click the **Application Files** button to open the **Application Files** dialog box.

4. In the **Application Files** dialog box, select the file that you wish to mark as data.
5. In the **Publish Status** field, select **Data File** from the drop-down list.

#### To mark files as prerequisites

1. With a project selected in **Solution Explorer**, on the **Project** menu, click **Properties**.
2. Click the **Publish** tab.
3. Click the **Application Files** button to open the **Application Files** dialog box.
4. In the **Application Files** dialog box, select the application assembly (.dll/file) that you wish to mark as a prerequisite. Note that your application must have a reference to the application assembly in order for it to appear in the list.
5. In the **Publish Status** field, select **Prerequisite** from the drop-down list.

#### To add a new file group

1. With a project selected in **Solution Explorer**, on the **Project** menu, click **Properties**.
2. Click the **Publish** tab.
3. Click the **Application Files** button to open the **Application Files** dialog box.
4. In the **Application Files** dialog box, select the **Group** field for a file that you wish to include in the new group.

#### NOTE

Files must have the **Build Action** property set to **Content** before the file names appear in the **Application Files** dialog box.

5. In the **Download Group** field, select **<New...>** from the drop-down list.
6. In the **New Group** dialog box, enter a name for the group, and then click **OK**.

#### To add a file to a group

1. With a project selected in **Solution Explorer**, on the **Project** menu, click **Properties**.
2. Click the **Publish** tab.
3. Click the **Application Files** button to open the **Application Files** dialog box.
4. In the **Application Files** dialog box, select the **Group** field for a file that you wish to include in the new group.
5. In the **Download Group** field, select a group from the drop-down list.

#### NOTE

You cannot change the **Download Group** for files that are required for the application to run.

## See also

- [Publish ClickOnce applications](#)
- [How to: Publish a ClickOnce application using the Publish Wizard](#)



# How to: Include a data file in a ClickOnce application

3/5/2021 • 2 minutes to read • [Edit Online](#)

Each ClickOnce application you install is assigned a data directory on the destination computer's local disk where the application can manage its own data. Data files can include files of any type: text files, XML files, or even Microsoft Access database (.mdb) files. The following procedures show you how to add a data file of any type into your ClickOnce application.

## To include a data file by using Mage.exe

1. Add the data file to your application directory with the rest of your application's files.

Typically, your application directory will be a directory labeled with the deployment's current version—for example, v1.0.0.0.

2. Update your application manifest to list the data file.

```
mage -u v1.0.0.0\Application.manifest -FromDirectory v1.0.0.0
```

Performing this task re-creates the list of files in your application manifest and also automatically generates the hash signatures.

3. Open the application manifest in your preferred text or XML editor and find the `file` element for your recently added file.

If you added an XML file named `Data.xml`, the file will look similar to the following code example.

```
<file name="Data.xml" hash="23454C18A2DC1D23E5B391FEE299B1F235067C59" hashalg="SHA1"
asmv2:size="39500" />
```

4. Add the attribute `type` to this element, and supply it with a value of `data`.

```
<file name="Data.xml" writeableType="applicationData" hash="23454C18A2DC1D23E5B391FEE299B1F235067C59"
hashalg="SHA1" asmv2:size="39500" />
```

5. Re-sign your application manifest by using your key pair or certificate, and then re-sign your deployment manifest.

You must re-sign your deployment manifest because its hash of the application manifest has changed.

```
mage -s app manifest -cf cert_file -pwd password
```

```
mage -u deployment manifest -appm app manifest
```

```
mage -s deployment manifest -cf certfile -pwd password
```

## To include a data file by using MageUI.exe

1. Add the data file to your application directory with the rest of your application's files.
2. Typically, your application directory will be a directory labeled with the deployment's current version—for example, v1.0.0.0.
3. On the **File** menu, click **Open** to open your application manifest.
4. Select the **Files** tab.
5. In the text box at the top of the tab, enter the directory that contains your application's files, and then click

### Populate.

Your data file will appear in the grid.

6. Set the **File Type** value of the data file to **Data**.
7. Save the application manifest, and then re-sign the file.

*MageUI.exe* will prompt you to re-sign the file.

8. Re-sign your deployment manifest

You must re-sign your deployment manifest because its hash of the application manifest has changed.

## See also

- [Access local and remote data in ClickOnce applications](#)

# How to: Install prerequisites with a ClickOnce application

3/5/2021 • 2 minutes to read • [Edit Online](#)

All ClickOnce applications require that the correct version of the .NET Framework is installed on a computer before they can be run; many applications have other prerequisites as well. When publishing a ClickOnce application, you can choose a set of prerequisite components to be packaged along with your application. At installation time, a check will be performed for each prerequisite to determine if it already exists; if not it will be installed prior to installing the ClickOnce application.

Instead of packaging and publishing prerequisites, you can also specify a download location for the components. For example, rather than including prerequisites with every application that you publish, you might use a centralized file share or Web location that contains the installers for all of your prerequisites—at install time, the components will be downloaded and installed from that location.

## IMPORTANT

You should add prerequisite installer packages to your development computer before you publish your first ClickOnce application. For more information, see [How to: Include Prerequisites with a ClickOnce Application](#).

Prerequisites are managed in the **Prerequisites** dialog box, accessible from the **Publish** pane of the **Project Designer**.

## NOTE

In addition to the predetermined list of prerequisites, you can add your own components to the list. For more information, see [Creating bootstrapper packages](#).

### To specify prerequisites to install with a ClickOnce application

1. With a project selected in **Solution Explorer**, on the **Project** menu click **Properties**.
2. Select the **Publish** pane.
3. Click the **Prerequisites** button to open the **Prerequisites** dialog box.
4. In the **Prerequisites** dialog box, make sure that the **Create setup program to install prerequisite components** check box is selected.
5. In the **Prerequisites** list, check the components that you wish to install, and then click **OK**.

The selected components will be packaged and published along with your application.

### To specify a different download location for prerequisites

1. With a project selected in **Solution Explorer**, on the **Project** menu click **Properties**.
2. Select the **Publish** pane.
3. Click the **Prerequisites** button to open the **Prerequisites** dialog box.
4. In the **Prerequisites** dialog box, make sure that the **Create setup program to install prerequisite components** check box is selected.

5. In the **Specify the install location for prerequisites** section, select **Download prerequisites from the following location**.
6. Select a location from the drop-down list, or enter a URL, file path, or FTP location, and then click **OK**.

**NOTE**

You must make sure that installers for the specified components exist at the specified location.

## See also

- [Publish ClickOnce applications](#)
- [How to: Publish a ClickOnce application using the Publish Wizard](#)

# How to: Include prerequisites with a ClickOnce application

3/5/2021 • 2 minutes to read • [Edit Online](#)

Before you can distribute prerequisite software with a ClickOnce application, you must first download the installer packages for those prerequisites to your development computer. When you publish an application and choose **Download prerequisites from the same location as my application**, an error will occur if the installer packages aren't in the **Packages** folder.

## NOTE

To add an installer package for the .NET Framework, see [.NET Framework Deployment Guide for Developers](#).

## To add an installer package by using Package.xml

1. In File Explorer, open the **Packages** folder.

By default, the path is `%ProgramFiles(x86)%\Microsoft SDKs\ClickOnce Bootstrapper\Packages\`.

2. Open the folder for the prerequisite that you want to add, and then open the language folder for your installed version of Visual Studio (for example, **en** for English).
3. In Notepad, open the *Package.xml* file.
4. Locate the **Name** element that contains `http://go.microsoft.com/fwlink`, and copy the URL. Include the **LinkID** portion.

## NOTE

If no **Name** element contains `http://go.microsoft.com/fwlink`, open the **Product.xml** file in the root folder for the prerequisite and locate the **fwlink** string.

## IMPORTANT

Some prerequisites have multiple installer packages (for example, for 32-bit or 64-bit systems). If multiple **Name** elements contain **fwlink**, you must repeat the remaining steps for each of them.

5. Paste the URL into the address bar of your browser, and then, when you are prompted to run or save, choose **Save**.

This step downloads the installer file to your computer.

6. Copy the file to the root folder for the prerequisite.

For example, for the Windows Installer 4.5 prerequisite, copy the file to the `\Packages\WindowsInstaller4_5` folder.

You can now distribute the installer package with your application.

## See also

- [How to: Install prerequisites with a ClickOnce application](#)

# How to: Manage updates for a ClickOnce application

3/5/2021 • 4 minutes to read • [Edit Online](#)

ClickOnce applications can check for updates automatically or programmatically. As a developer, you have lots of flexibility in specifying when and how update checks are performed, whether updates are mandatory, and where the application should check for updates.

You can configure the application to check for updates automatically before the application starts, or at set intervals after the application starts. In addition you can specify a minimum required version; that is, an update is installed if the user's version is lower than the required version.

You can configure the application to check for updates programmatically based on an event such as a user request. The procedure "To check for updates programmatically" in this topic shows how you would write code that uses the [ApplicationDeployment](#) class to check for updates based on an event.

You can also deploy your application from one location and update it from another. See the procedure "To specify a different update location."

For more information, see [Choosing a ClickOnce Update Strategy](#).

Update behavior is managed in the **Application Updates** dialog box, available from the **Publish** page of the **Project Designer**.

## To check for updates before the application starts

1. With a project selected in **Solution Explorer**, on the **Project** menu, click **Properties**.
2. Click the **Publish** tab.
3. Click the **Updates** button to open the **Application Updates** dialog box.
4. In the **Application Updates** dialog box, make sure that the **The application should check for updates** check box is selected.
5. In the **Choose when the application should check for updates** section, select **Before the application starts**. This ensures that users connected to the network always run the application with the latest updates.

## To check for updates in the background after the application starts

1. With a project selected in **Solution Explorer**, on the **Project** menu, click **Properties**.
2. Click the **Publish** tab.
3. Click the **Updates** button to open the **Application Updates** dialog box.
4. In the **Application Updates** dialog box, make sure that the check box **The application should check for updates** is selected.
5. In the **Choose when the application should check for updates** section, select **After the application starts**. The application will start more quickly this way, and then it will check for updates in the background, and only notify the user when an update is available. Once installed, updates will not take effect until the application is restarted.
6. In the **Specify how frequently the application should check for updates** section, select either

Check every time the application runs (the default) or **Check every** and enter a number and time interval.

#### To specify a minimum required version for the application

1. With a project selected in **Solution Explorer**, on the **Project** menu, click **Properties**.
2. Click the **Publish** tab.
3. Click the **Updates** button to open the **Application Updates** dialog box.
4. In the **Application Updates** dialog box, make sure that the **The application should check for updates** check box is selected.
5. Select the **Specify a minimum required version for this application** check box, and then enter **Major**, **Minor**, **Build**, and **Revision** numbers for the application.

#### To specify a different update location

1. With a project selected in **Solution Explorer**, on the **Project** menu, click **Properties**.
2. Click the **Publish** tab.
3. Click the **Updates** button to open the **Application Updates** dialog box.
4. In the **Application Updates** dialog box, make sure that the **The application should check for updates** check box is selected.
5. In the **Update location** field, enter the update location with a fully qualified URL, using the format *http://Hostname/ApplicationName*, or a UNC path using the format *\\Server\ApplicationName*, or click the **Browse** button to browse for the update location.

#### To check for updates programmatically

1. With a project selected in **Solution Explorer**, on the **Project** menu, click **Properties**.
2. Click the **Publish** tab.
3. Click the **Updates** button to open the **Application Updates** dialog box.
4. In the **Application Updates** dialog box, make sure that the **The application should check for updates** check box is cleared. (Optionally, you can select this check box to check for updates programmatically and also let the ClickOnce runtime check for updates automatically.)
5. In the **Update location** field, enter the update location with a fully qualified URL, using the format *http://Hostname/ApplicationName*, or a UNC path using the format *\\Server\ApplicationName*, or click the **Browse** button to browse for the update location. The update location is where the application will look for an updated version of itself.
6. Create a button, menu item, or other user interface item on a Windows Form that users will select to check for updates. From that item's event handler, call a method to check for and install updates. You can find an example of Visual Basic and Visual C# code for such a method in [How to: Check for application updates programmatically using the ClickOnce deployment API](#).
7. Build your application.

## See also

- [ApplicationDeployment](#)
- [Application updates dialog box](#)
- [Choose a ClickOnce update strategy](#)
- [Publish ClickOnce applications](#)



- [How to: Publish a ClickOnce application using the Publish Wizard](#)
- [How to: Check for application updates programmatically using the ClickOnce deployment API](#)

# How to: Change the publish language for a ClickOnce application

3/5/2021 • 2 minutes to read • [Edit Online](#)

When publishing a ClickOnce application, the user interface displayed during installation defaults to the language and culture of your development computer. If you are publishing a localized application, you will need to specify a language and culture to match the localized version. This is determined by the `Publish language` property for your project.

The `Publish language` property can be set in the **Publish Options** dialog box, accessible from the **Publish** page of the **Project Designer**.

## NOTE

The dialog boxes and menu commands you see might differ from those described in Help depending on your active settings or edition. To change your settings, choose **Import and Export Settings** on the **Tools** menu. For more information, see [Reset settings](#).

## To change the publish language

1. With a project selected in **Solution Explorer**, on the **Project** menu, click **Properties**.
2. Click the **Publish** tab.
3. Click the **Options** button to open the **Publish Options** dialog box.
4. Click **Description**.
5. In the **Publish Options** dialog box, select a language and culture from the **Publish language** drop-down list, and then click **OK**.

## See also

- [Publish ClickOnce applications](#)
- [How to: Publish a ClickOnce application using the Publish Wizard](#)

# How to: Specify a Start menu name for a ClickOnce application

3/5/2021 • 2 minutes to read • [Edit Online](#)

When a ClickOnce application is installed for both online and offline use, an entry is added to the **Start** menu and the **Add or Remove Programs** list. By default, the display name is the same as the name of the application assembly, but you can change the display name by setting **Product name** in the **Publish Options** dialog box.

**Product name** will be displayed on the *publish.htm* page; for an installed offline application, it will be the name of the entry in the **Start** menu, and it will also be the name that shows in **Add or Remove Programs**.

**Publisher name** will appear on the *publish.htm* page above **Product name**, and for an installed offline application, it will also be the name of the folder that contains the application's icon in the **Start** menu.

The Start menu shortcut or app reference gets created in *%appdata%\Microsoft\Windows\Start Menu\Programs\<publisher name>*. The shortcut or app reference has the same name as the product name.

You can set the **Product name** and **Publisher name** properties in the **Publish Options** dialog box, available on the **Publish** page of the **Project Designer**.

## To specify a Start menu name

1. With a project selected in **Solution Explorer**, on the **Project** menu, click **Properties**.
2. Click the **Publish** tab.
3. Click the **Options** button to open the **Publish Options** dialog box.
4. Click **Description**.
5. In the **Publish Options** dialog box, enter the name to display in **Product name**.
6. Optionally, you can enter a publisher name in **Publisher name**.

## See also

- [Publish ClickOnce applications](#)
- [How to: Publish a ClickOnce application using the Publish Wizard](#)

# How to: Specify a link for Technical Support

3/5/2021 • 2 minutes to read • [Edit Online](#)

When publishing a ClickOnce application, the **Support URL** property identifies a Web page or file share where users can go to get information about the application. This property is optional; if provided, the URL will be displayed in the application's entry **Add or Remove Programs** dialog box.

The **Support URL** property can be set on the **Publish** page of the **Project Designer**.

## To specify a support URL

1. With a project selected in **Solution Explorer**, on the **Project** menu, click **Properties**.
2. Click the **Publish** tab.
3. Click the **Options** button to open the **Publish Options** dialog box.
4. Click **Description**.
5. In the **Support URL** field, enter a fully qualified path to a Web site, Web page, or UNC share.

## See also

- [Publish ClickOnce applications](#)
- [How to: Publish a ClickOnce application using the Publish Wizard](#)

# How to: Specify a support URL for individual prerequisites in a ClickOnce deployment

3/5/2021 • 2 minutes to read • [Edit Online](#)

A ClickOnce deployment can test for a number of prerequisites that must be available on the client computer for the ClickOnce application to run. These dependencies include the required minimum version of the .NET Framework, the version of the operating system, and any assemblies that must be preinstalled in the global assembly cache (GAC). ClickOnce, however, cannot install any of these prerequisites itself; if a prerequisite is not found, it simply halts installation and displays a dialog box explaining why the installation failed.

There are two methods for installing prerequisites. You can install them using a bootstrapper application. Alternatively, you can specify a support URL for individual prerequisites, which is displayed to users on the dialog box if the prerequisite is not found. The page referenced by that URL can contain links to instructions for installing the required prerequisite. If an application does not specify a support URL for an individual prerequisite, ClickOnce displays the support URL specified in the deployment manifest for the application as a whole, if it is defined.

While Visual Studio, *Mage.exe*, and *MageUI.exe* can all be used to generate ClickOnce deployments, none of these tools directly support specifying a support URL for individual prerequisites. This document describes how to modify your deployment's application manifest and deployment manifest to include these support URLs.

## Specify a support URL for an individual prerequisite

1. Open the application manifest (the *.manifest* file) for the ClickOnce application in a text editor.
2. For an operating system prerequisite, add the `supportUrl` attribute to the `dependentOS` element:

```
<dependency>
  <dependentOS supportUrl="http://www.adatum.com/MyApplication/wrongOSFound.htm">
    <osVersionInfo>
      <os majorVersion="5" minorVersion="1" buildNumber="2600" servicePackMajor="0"
servicePackMinor="0" />
    </osVersionInfo>
  </dependentOS>
</dependency>
```

3. For a prerequisite for a certain version of the common language runtime, add the `supportUrl` attribute to the `dependentAssembly` entry that specifies the common language runtime dependency:

```
<dependency>
  <dependentAssembly dependencyType="preRequisite" allowDelayedBinding="true" supportUrl="
http://www.adatum.com/MyApplication/wrongClrVersionFound.htm">
    <assemblyIdentity name="Microsoft.Windows.CommonLanguageRuntime" version="4.0.30319.0" />
  </dependentAssembly>
</dependency>
```

4. For a prerequisite for an assembly that must be preinstalled in the global assembly cache, set the `supportUrl` for the `dependentAssembly` element that specifies the required assembly:

```
<dependency>
  <dependentAssembly dependencyType="preRequisite" allowDelayedBinding="true" supportUrl="
http://www.adatum.com/MyApplication/missingSampleGACAssembly.htm">
    <assemblyIdentity name="SampleGACAssembly" version="5.0.0.0" publicKeyToken="04529dfb5da245c5"
processorArchitecture="msil" language="neutral" />
  </dependentAssembly>
</dependency>
```

5. Optional. For applications that target the .NET Framework 4, open the deployment manifest (the *.application* file) for the ClickOnce application in a text editor.
6. For a .NET Framework 4 prerequisite, add the `supportUrl` attribute to the `compatibleFrameworks` element:

```
<compatibleFrameworks xmlns="urn:schemas-microsoft-com:clickonce.v2"
supportUrl="http://adatum.com/MyApplication/CompatibleFrameworks.htm">
  <framework targetVersion="4.0" profile="Client" supportedRuntime="4.0.30319" />
  <framework targetVersion="4.0" profile="Full" supportedRuntime="4.0.30319" />
</compatibleFrameworks>
```

7. Once you have manually altered the application manifest, you must re-sign the application manifest using your digital certificate, then update and re-sign the deployment manifest as well. Use the *Mage.exe* or *MageUI.exe* SDK tools to accomplish this task, as regenerating these files using Visual Studio erases your manual changes. For more information on using Mage.exe to re-sign manifests, see [How to: Re-sign Application and Deployment Manifests](#).

## .NET Framework security

The support URL is not displayed on the dialog box if the application is marked to run in partial trust.

## See also

- [Mage.exe \(Manifest Generation and Editing Tool\)](#)
- [Walkthrough: Manually deploy a ClickOnce application](#)
- `<compatibleFrameworks>` element
- [ClickOnce and Authenticode](#)
- [Application deployment prerequisites](#)

# How to: Specify a publish page for a ClickOnce application

3/5/2021 • 2 minutes to read • [Edit Online](#)

When publishing a ClickOnce application, a default Web page (publish.htm) is generated and published along with the application. This page contains the name of the application, a link to install the application and/or any prerequisites, and a link to a Help topic describing ClickOnce. The **Publish Page** property for your project allows you to specify a name for the Web page for your ClickOnce application.

Once the publish page has been specified, the next time you publish, it will be copied to the publish location; it will not be overwritten if you publish again. If you wish to customize the appearance of the page, you can do so without worrying about losing your changes. For more information, see [How to: Customize the ClickOnce default Web page](#).

The **Publish Page** property can be set in the **Publish Options** dialog box, accessible from the **Publish** pane of the **Project Designer**.

## To specify a custom Web page for a ClickOnce application

1. With a project selected in **Solution Explorer**, on the **Project** menu click **Properties**.
2. Select the **Publish** pane.
3. Click the **Options** button to open the **Publish Options** dialog box.
4. Click **Deployment**.
5. In the **Publish Options** dialog box, make sure that the **Open deployment web page after publish** check box is selected (it should be selected by default).
6. In the **Deployment web page** box, enter the name for your Web page, and then click **OK**.

## To prevent the publish page from launching each time you publish

1. With a project selected in **Solution Explorer**, on the **Project** menu click **Properties**.
2. Select the **Publish** pane.
3. Click the **Options** button to open the **Publish Options** dialog box.
4. Click **Deployment**.
5. In the **Publish Options** dialog box, clear the **Open deployment web page after publish** check box.

## See also

- [Publish ClickOnce applications](#)
- [How to: Publish a ClickOnce application using the Publish Wizard](#)
- [How to: Customize the ClickOnce default Web page](#)

# How to: Customize the default Web page for a ClickOnce application

3/5/2021 • 2 minutes to read • [Edit Online](#)

When publishing a ClickOnce application to the Web, a Web page is automatically generated and published along with the application. The default page contains the name of the application and links to install the application, install prerequisites, or access help on MSDN.

## NOTE

The actual links that you see on the page depend on the computer where the page is being viewed and what prerequisites you are including.

The default name for the Web page is *Publish.htm*; you can change the name in the **Project Designer**. For more information, see [How to: Specify a publish page for a ClickOnce application](#).

The *Publish.htm* Web page is published only if a newer version is detected.

## NOTE

Changes that you make to your **Publish** settings will not affect the *Publish.htm* page, with one exception: if you add or remove prerequisites after initially publishing, the list of prerequisites will no longer be accurate. You will need to edit the text for the prerequisite link to reflect the changes.

## To customize the publish Web page

1. Publish your ClickOnce application to a Web location. For more information, see [How to: Publish a ClickOnce application using the Publish Wizard](#).
2. On the Web server, open the *Publish.htm* file in Visual Web Designer or another HTML editor.
3. Customize the page as desired and save it.
4. Optional. To prevent Visual Studio from overwriting your customized publish Web page, uncheck **Automatically generate deployment Web page after every publish** in the **Publish Options** dialog box.

## See also

- [ClickOnce security and deployment](#)
- [Publishing ClickOnce applications](#)
- [How to: Install prerequisites with a ClickOnce application](#)
- [How to: Specify a publish page for a ClickOnce application](#)



# How to: Enable AutoStart for CD installations

3/5/2021 • 2 minutes to read • [Edit Online](#)

When deploying a ClickOnce application by means of removable media such as CD-ROM or DVD-ROM, you can enable `AutoStart` so that the ClickOnce application is automatically launched when the media is inserted.

`AutoStart` can be enabled on the **Publish** page of the **Project Designer**.

## To enable AutoStart

1. With a project selected in **Solution Explorer**, on the **Project** menu click **Properties**.
2. Click the **Publish** tab.
3. Click the **Options** button.

The **Publish Options** dialog box appears.

4. Click **Deployment**.
5. Select the **For CD installations, automatically start Setup when CD is inserted** check box.

An *Autorun.inf* file will be copied to the publish location when the application is published.

## See also

- [Publish ClickOnce applications](#)
- [How to: Publish a ClickOnce application using the Publish Wizard](#)

# How to: Create file associations for a ClickOnce application

3/5/2021 • 2 minutes to read • [Edit Online](#)

ClickOnce applications can be associated with one or more file name extensions, so that the application will be started automatically when the user opens a file of those types. Adding file name extension support to a ClickOnce application is straightforward.

## To create file associations for a ClickOnce application

1. Create a ClickOnce application normally, or use your existing ClickOnce application.
2. Open the application manifest with a text or XML editor, such as Notepad.
3. Find the `assembly` element. For more information, see [ClickOnce application manifest](#).
4. As a child of the `assembly` element, add a `fileAssociation` element. The `fileAssociation` element has four attributes:
  - `extension` : The file name extension you want to associate with the application.
  - `description` : A description of the file type, which will appear in the Windows shell.
  - `progid` : A string uniquely identifying the file type, to mark it in the registry.
  - `defaultIcon` : An icon to use for this file type. The icon must be added as a file resource in the application manifest. For more information, see [How to: Include a Data File in a ClickOnce Application](#).

For an example of the `file` and `fileAssociation` elements, see [<fileAssociation> Element](#).

5. If you want to associate more than one file type with the application, add additional `fileAssociation` elements. Note that the `progid` attribute should be different for each.
6. Once you have finished with the application manifest, re-sign the manifest. You can do this from the command line by using *Mage.exe*.

```
mage -Sign WindowsFormsApp1.exe.manifest -CertFile mycert.pfx
```

For more information, see [Mage.exe \(Manifest Generation and Editing Tool\)](#).

## See also

- [<fileAssociation> element](#)
- [ClickOnce application manifest](#)
- [Mage.exe \(Manifest Generation and Editing Tool\)](#)

# How to: Retrieve query string information in an online ClickOnce application

4/2/2021 • 3 minutes to read • [Edit Online](#)

The *query string* is the portion of a URL beginning with a question mark (?) that contains arbitrary information in the form *name=value*. Suppose you have a ClickOnce application named `WindowsApp1` that you host on `servername`, and you want to pass in a value for the variable `username` when the application launches. Your URL might look like the following:

```
http://servername/WindowsApp1.application?username=joeuser
```

The following two procedures show how to use a ClickOnce application to obtain query string information.

## NOTE

You can only pass information in a query string when your application is being launched using HTTP, instead of using a file share or the local file system.

The first procedure shows how your ClickOnce application can use a small piece of code to read these values when the application launches.

The next procedure shows how to configure your ClickOnce application using `MageUI.exe` so that it can accept query string parameters. You will need to do this whenever you publish your application.

## NOTE

See the "Security" section later in this topic before you make a decision to enable this feature.

For information about how to create a ClickOnce deployment using *Mage.exe* or *MageUI.exe*, see [Walkthrough: Manually deploy a ClickOnce application](#).

## NOTE

Starting in .NET Framework 3.5 SP1, it is possible to pass command-line arguments to an offline ClickOnce application. If you want to supply arguments to the application, you can pass in parameters to the shortcut file with the .APPREF-MS extension.

## To obtain query string information from a ClickOnce application

1. Place the following code in your project. In order for this code to function, you will have to have a reference to `System.Web` and add `using` or `Imports` directives for `System.Web`, `System.Collections.Specialized`, and `System.Deployment.Application`.

```
private NameValueCollection GetQueryStringParameters()
{
    NameValueCollection nameValueCollection = new NameValueCollection();

    if (ApplicationDeployment.IsNetworkDeployed)
    {
        string queryString = ApplicationDeployment.CurrentDeployment.ActivationUri.Query;
        nameValueCollection = HttpUtility.ParseQueryString(queryString);
    }

    return (nameValueCollection);
}
```

```
Private Function GetQueryStringParameters() As NameValueCollection
    Dim NameValueCollection As New NameValueCollection()

    If (ApplicationDeployment.IsNetworkDeployed) Then
        Dim QueryString As String = ApplicationDeployment.CurrentDeployment.ActivationUri.Query
        NameValueCollection = HttpUtility.ParseQueryString(QueryString)
    End If

    GetQueryStringParameters = NameValueCollection
End Function
```

2. Call the function defined previously to retrieve a [Dictionary](#) of the query string parameters, indexed by name.

### To enable query string passing in a ClickOnce application with MageUI.exe

1. Open the .NET Command Prompt and type:

```
MageUI
```

2. From the **File** menu, select **Open**, and open the deployment manifest for your ClickOnce application, which is the file ending in the `.application` extension.
3. Select the **Deployment Options** panel in the left-hand navigation window, and select the **Allow URL parameters to be passed to application** check box.
4. From the **File** menu, select **Save**.

#### NOTE

Alternately, you can enable query string passing in Visual Studio. Select the **Allow URL parameters to be passed to application** check box, which can be found by opening the **Project Properties**, selecting the **Publish** tab, clicking the **Options** button, and then selecting **Manifests**.

## Robust programming

When you use query string parameters, you must give careful consideration to how your application is installed and activated. If your application is configured to install on the user's computer from the Web or from a network share, it is likely that the user will activate the application only once through the URL. After that, the user will usually activate your application using the shortcut in the **Start** menu. As a result, your application is guaranteed to receive query string arguments only once during its lifetime. If you choose to store these arguments on the user's machine for future use, you are responsible for storing them in a safe and secure manner.

If your application is online only, it will always be activated through a URL. Even in this case, however, your application must be written to function properly if the query string parameters are missing or corrupted.

## .NET Framework security

Allow passing URL parameters to your ClickOnce application only if you plan to cleanse the input of any malicious characters before using it. A string embedded with quotes, slashes, or semicolons, for example, might perform arbitrary data operations if used unfiltered in a SQL query against a database. For more information on query string security, see [Script exploits overview](#).

## See also

- [Secure ClickOnce applications](#)

# How to: Disable URL activation of ClickOnce applications by using the Designer

3/5/2021 • 2 minutes to read • [Edit Online](#)

Typically, a ClickOnce application will start automatically immediately after it is installed from a Web server. For security reasons, you may decide to disable this behavior, and tell users to start the application from the **Start** menu instead. The following procedure describes how to disable URL activation.

This technique can be used only for ClickOnce applications installed on the user's computer from a Web server. It cannot be used for online-only applications, which can be started only by using their URL. For more information about the difference between online-only and installed applications, see [Choosing a ClickOnce Deployment Strategy](#).

This procedure uses Visual Studio. You can also accomplish this task by using the Windows Software Development Kit (SDK). For more information, see [How to: Disable URL Activation of ClickOnce Applications](#).

## Procedure

**To disable URL activation for your application**

1. Right-click your project name in **Solution Explorer**, and click **Properties**.
2. On the **Properties** page, click the **Publish** tab.
3. Click **Options**.
4. Click **Manifests**.
5. Select the check box labeled **Block application from being activated via a URL**.
6. Deploy your application.

## See also

- [Publishing ClickOnce applications](#)

# How to: Disable URL activation of ClickOnce applications

3/5/2021 • 2 minutes to read • [Edit Online](#)

Typically, a ClickOnce application will launch automatically immediately after it is installed from a Web server. For security reasons, you may decide to disable this behavior, and tell users to launch the application from the **Start** menu instead. The following procedure describes how to disable URL activation.

This technique can be used only for ClickOnce applications installed on the user's computer from a Web server. It cannot be used for online-only applications, which can be launched only by using their URL. For more information on the difference between online-only and installed applications, see [Choosing a ClickOnce Deployment Strategy](#).

This procedure uses the Windows Software Development Kit (SDK) tool MageUI.exe. For more information on this tool, see [MageUI.exe \(Manifest Generation and Editing Tool, Graphical Client\)](#). You can also perform this procedure using Visual Studio.

## Procedure

### To disable URL activation for your application

1. Open your deployment manifest in MageUI.exe. If you have not yet created one, follow the steps in [Walkthrough: Manually deploy a ClickOnce application](#).
2. Select the **Deployment Options** tab.
3. Clear the **Automatically run application after installing** check box.
4. Save and sign the manifest.

## See also

- [Publish ClickOnce applications](#)

# How to: Use ClickOnce to deploy applications that can run on multiple versions of the .NET framework

3/5/2021 • 2 minutes to read • [Edit Online](#)

You can deploy an application that targets multiple versions of the .NET Framework by using the ClickOnce deployment technology. This requires that you generate and update the application and deployment manifests.

## NOTE

Before you change the application to target multiple versions of the .NET Framework, you should ensure that your application runs with multiple versions of the .NET Framework. The version common language runtime is different between .NET Framework 4 versus .NET Framework 2.0, .NET Framework 3.0, and .NET Framework 3.5.

This process requires the following steps:

1. Generate the application and deployment manifests.
2. Change the deployment manifest to list the multiple .NET Framework versions.
3. Change the *app.config* file to list the compatible .NET Framework runtime versions.
4. Change the application manifest to mark dependent assemblies as .NET Framework assemblies.
5. Sign the application manifest.
6. Update and sign the deployment manifest.

## To generate the application and deployment manifests

- Use the Publish Wizard or the Publish Page of the Project Designer to publish the application and generate the application and deployment manifest files. For more information, see [How to: Publish a ClickOnce application using the Publish Wizard](#) or [Publish Page, Project Designer](#).

## To change the deployment manifest to list the multiple .NET Framework versions

1. In the publish directory, open the deployment manifest by using the XML Editor in Visual Studio. The deployment manifest has the *.application* file name extension.
2. Replace the XML code between the

`<compatibleFrameworks xmlns="urn:schemas-microsoft-com:clickonce.v2">` and `</compatibleFrameworks>`

elements with XML that lists the supported .NET Framework versions for your application.

The following table shows some of the available .NET Framework versions and the corresponding XML that you can add to the deployment manifest.

| .NET FRAMEWORK VERSION | XML  |
|------------------------|--|
| 4 Client               | <code>&lt;framework targetVersion="4.0" profile="Client" supportedRuntime="4.0.30319" /&gt;</code> |
| 4 Full                 | <code>&lt;framework targetVersion="4.0" profile="Full" supportedRuntime="4.0.30319" /&gt;</code>   |



| .NET Framework Version | XML   |
|------------------------|---|
| 3.5 Client             | <framework targetVersion="3.5" profile="Client" supportedRuntime="2.0.50727" /> |
| 3.5 Full               | <framework targetVersion="3.5" profile="Full" supportedRuntime="2.0.50727" />   |
| 3.0                    | <framework targetVersion="3.0" supportedRuntime="2.0.50727" />                  |

### To change the app.config file to list the compatible .NET Framework runtime versions

1. In Solution Explorer, open the *app.config* file by using the XML Editor in Visual Studio.
2. Replace (or add) the XML code between the <startup> and </startup> elements with XML that lists the supported .NET Framework runtimes for your application.

The following table shows some of the available .NET Framework versions and the corresponding XML that you can add to the deployment manifest.

| .NET Framework Runtime Version | XML   |
|--------------------------------|---|
| 4 Client                       | <supportedRuntime version="v4.0.30319" sku=".NETFramework,Version=v4.0,Profile=Client" /> |
| 4 Full                         | <supportedRuntime version="v4.0.30319" sku=".NETFramework,Version=v4.0" />                |
| 3.5 Full                       | <supportedRuntime version="v2.0.50727"/>  |
| 3.5 Client                     | <supportedRuntime version="v2.0.50727" sku="Client"/>                                     |

### To change the application manifest to mark dependent assemblies as .NET Framework assemblies

1. In the publish directory, open the application manifest by using the XML Editor in Visual Studio. The deployment manifest has the *.manifest* file name extension.
2. Add group="framework" to the dependency XML for the sentinel assemblies ( System.Core , WindowsBase , Sentinel.v3.5Client , and System.Data.Entity ). For example, the XML should look like the following:

```
<dependentAssembly dependencyType="preRequisite" allowDelayedBinding="true" group="framework">
```

3. Update the version number of the <assemblyIdentity> element for Microsoft.Windows.CommonLanguageRuntime to the version number for the .NET Framework that is the lowest common denominator. For example, if the application targets .NET Framework 3.5 and .NET Framework 4, use the 2.0.50727.0 version number and the XML should look like the following:

```
<dependency>
  <dependentAssembly dependencyType="preRequisite" allowDelayedBinding="true">
    <assemblyIdentity name="Microsoft.Windows.CommonLanguageRuntime" version="2.0.50727.0" />
  </dependentAssembly>
</dependency>
```

### To update and re-sign the application and deployment manifests

- Update and re-sign the application and deployment manifests. For more information, see [How to: Re-sign application and deployment manifests](#).

## See also

- [Publish ClickOnce applications](#)
- [<compatibleFrameworks> element](#)
- [<dependency> element](#)
- [ClickOnce deployment manifest](#)
- [Configuration file schema](#)

# How to: Publish a WPF application with visual styles enabled

3/5/2021 • 5 minutes to read • [Edit Online](#)

Visual styles enable the appearance of common controls to change based on the theme chosen by the user. By default, visual styles are not enabled for Windows Presentation Foundation (WPF) applications, so you must enable them manually. However, enabling visual styles for a WPF application and then publishing the solution causes an error. This topic describes how to resolve this error and the process for publishing a WPF application with visual styles enabled. For more information about visual styles, see [Visual styles overview](#). For more information about the error message, see [Troubleshoot specific errors in ClickOnce deployments](#).

To resolve the error and to publish the solution, you must perform the following tasks:

- [Publish the solution without visual styles enabled](#).
- [Create a manifest file](#).
- [Embed the manifest file into the executable file of the published solution](#).
- [Sign the application and deployment manifests](#).

Then, you can move the published files to the location from which you want end users to install the application.

## Publish the solution without visual styles enabled

1. Ensure that your project does not have visual styles enabled. First, check your project's manifest file for the following XML. Then, if the XML is present, enclose the XML with a comment tag.

By default, visual styles are not enabled.

```
<dependency>
  <dependentAssembly>
    <assemblyIdentity type="win32" name="Microsoft.Windows.Common-Controls" version="6.0.0.0"
processorArchitecture="*" publicKeyToken="6595b64144ccf1df" language="*" />
  </dependentAssembly>
</dependency>
```

The following procedures show how to open the manifest file associated with your project.

### To open the manifest file in a Visual Basic project

- a. On the menu bar, choose **Project, *ProjectName* Properties**, where *ProjectName* is the name of your WPF project.

The property pages for your WPF project appear.

- b. On the **Application** tab, choose **View Windows Settings**.

The app.manifest file opens in the **Code Editor**.

### To open the manifest file in a C# project

- a. On the menu bar, choose **Project, *ProjectName* Properties**, where *ProjectName* is the name of your WPF project.

The property pages for your WPF project appear.

- b. On the **Application** tab, make a note of the name that appears in the manifest field. This is the name of the manifest that is associated with your project.

#### NOTE

If **Embed manifest with default settings** or **Create application without manifest** appear in the manifest field, visual styles are not enabled. If the name of a manifest file appears in the manifest field, proceed to the next step in this procedure.

- c. In **Solution Explorer**, choose **Show All Files**.

This button shows all project items, including those that have been excluded and those that are normally hidden. The manifest file appears as a project item.

2. Build and publish your solution. For more information about how to publish the solution, see [How to: Publish a ClickOnce application using the Publish Wizard](#).

## Create a manifest file

1. Paste the following XML into a Notepad file.

This XML describes the assembly that contains controls that support visual styles.

```
<?xml version="1.0" encoding="utf-8"?>
<asmv1:assembly manifestVersion="1.0"
  xmlns="urn:schemas-microsoft-com:asm.v1"
  xmlns:asmv1="urn:schemas-microsoft-com:asm.v1"
  xmlns:asmv2="urn:schemas-microsoft-com:asm.v2"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <dependency>
    <dependentAssembly>
      <assemblyIdentity type="win32" name="Microsoft.Windows.Common-Controls" version="6.0.0.0"
        processorArchitecture="*" publicKeyToken="6595b64144ccf1df" language="*" />
    </dependentAssembly>
  </dependency>
</asmv1:assembly>
```

2. In Notepad, click **File**, and then click **Save As**.
3. In the **Save As** dialog box, in the **Save as type** drop-down list, select **All Files**.
4. In the **File name** box, name the file and append *.manifest* to the end of the file name. For example: *themes.manifest*.
5. Choose the **Browse Folders** button, select any folder, and then click **Save**.

#### NOTE

The remaining procedures assume that the name of this file is *themes.manifest* and that the file is saved to the *C:\temp* directory on your computer.

## Embed the manifest file into the executable file of the published solution

1. Open **Developer Command Prompt for Visual Studio**.

For more information about how to open Developer Command Prompt for Visual Studio, see [Developer Command Prompt and Developer PowerShell](#).

#### NOTE

The remaining steps make the following assumptions about your solution:

- The name of the solution is **MyWPFFProject**.
- The solution is located in the following directory:  
`%UserProfile%\Documents\Visual Studio 2010\Projects\.`
- The solution is published to the following directory:  
`%UserProfile%\Documents\Visual Studio 2010\Projects\publish.`
- The most recent version of the published application files is located in the following directory:  
`%UserProfile%\Documents\Visual Studio 2010\Projects\publish\Application Files\WPFAApp_1_0_0_0`

You do not have to use the name or the directory locations described above. The name and locations described above are used only to illustrate the steps required to publish your solution.

2. At the command prompt, change the path to the directory that contains the most recent version of the published application files. The following example demonstrates this step.

```
cd "%UserProfile%\Documents\Visual Studio 2010\Projects\MyWPFFProject\publish\Application Files\WPFAApp_1_0_0_0"
```

3. At the command prompt, run the following command to embed the manifest file into the executable file of the application.

```
mt -manifest c:\temp\themes.manifest -outputresource:MyWPFAApp.exe.deploy
```

## Sign the application and deployment manifests

1. At the command prompt, run the following command to remove the *.deploy* extension from the executable file in the current directory.

```
ren MyWPFAApp.exe.deploy MyWPFAApp.exe
```

#### NOTE

This example assumes that only one file has the *.deploy* file extension. Make sure that you rename all files in this directory that have the *.deploy* file extension.

2. At the command prompt, run the following command to sign the application manifest.

```
mage -u MyWPFAApp.exe.manifest -cf ..\..\..\MyWPFAApp_TemporaryKey.pfx
```

#### NOTE

This example assumes that you sign the manifest by using the *.pfx* file of the project. If you are not signing the manifest, you can omit the `-cf` parameter that is used in this example. If you are signing the manifest with a certificate that requires a password, specify the `-password` option (

```
For example: mage -u MyWPFAApp.exe.manifest -cf ..\..\..\MyWPFAApp_TemporaryKey.pfx - password Password
```

).

3. At the command prompt, run the following command to add the *.deploy* extension to the name of the file that you renamed in a previous step of this procedure.

```
ren MyWPFAApp.exe MyWPFAApp.exe.deploy
```

#### NOTE

This example assumes that only one file had a *.deploy* file extension. Make sure that you rename all files in this directory that previously had the *.deploy* file name extension.

4. At the command prompt, run the following command to sign the deployment manifest.

```
mage -u ..\..\MyWPFAApp.application -appm MyWPFAApp.exe.manifest -cf ..\..\..\MyWPFAApp_TemporaryKey.pfx
```

#### NOTE

This example assumes that you sign the manifest by using the *.pfx* file of the project. If you are not signing the manifest, you can omit the `-cf` parameter that is used in this example. If you are signing the manifest with a certificate that requires a password, specify the `-password` option, as in this example:

```
For example: mage -u MyWPFAApp.exe.manifest -cf ..\..\..\MyWPFAApp_TemporaryKey.pfx - password Password
```

After you have performed these steps, you can move the published files to the location from which you want end users to install the application. If you intend to update the solution often, you can move these commands into a script and run the script each time that you publish a new version.

## See also

- [Troubleshooting Specific Errors in ClickOnce Deployments](#)
- [Visual Styles Overview](#)
- [Enabling Visual Styles](#)
- [Developer Command Prompt and Developer PowerShell](#)

# Walkthrough: Download assemblies on demand with the ClickOnce deployment API using the Designer

4/2/2021 • 6 minutes to read • [Edit Online](#)

By default, all the assemblies included in a ClickOnce application are downloaded when the application is first run. However, there might be parts of your application that are used by a small set of the users. In this case, you want to download an assembly only when you create one of its types. The following walkthrough demonstrates how to mark certain assemblies in your application as "optional", and how to download them by using classes in the [System.Deployment.Application](#) namespace when the common language runtime demands them.

## NOTE

Your application will have to run in full trust to use this procedure.

## NOTE

The dialog boxes and menu commands you see might differ from those described in Help depending on your active settings or edition. To change your settings, click **Import and Export Settings** on the **Tools** menu. For more information, see [Reset settings](#).

## Create the projects

### To create a project that uses an on-demand assembly with Visual Studio

1. Create a new Windows Forms project in Visual Studio. On the **File** menu, point to **Add**, and then click **New Project**. Choose a **Class Library** project in the dialog box and name it `ClickOnceLibrary`.

## NOTE

In Visual Basic, we recommend that you modify the project properties to change the root namespace for this project to `Microsoft.Samples.ClickOnceOnDemand` or to a namespace of your choice. For simplicity, the two projects in this walkthrough are in the same namespace.

2. Define a class named `DynamicClass` with a single property named `Message`.

```
Public Class DynamicClass
    Sub New()

    End Sub

    Public ReadOnly Property Message() As String
        Get
            Message = "Hello, world!"
        End Get
    End Property
End Class
```

```

using System;
using System.Collections.Generic;
using System.Text;

namespace Microsoft.Samples.ClickOnceOnDemand
{
    public class DynamicClass
    {
        public DynamicClass() {}

        public string Message
        {
            get
            {
                return ("Hello, world!");
            }
        }
    }
}

```

3. Select the Windows Forms project in **Solution Explorer**. Add a reference to the [System.Deployment.Application](#) assembly and a project reference to the `ClickOnceLibrary` project.

#### NOTE

In Visual Basic, we recommend that you modify the project properties to change the root namespace for this project to `Microsoft.Samples.ClickOnceOnDemand` or to a namespace of your choice. For simplicity, the two projects in this walkthrough are located in the same namespace.

4. Right-click the form, click **View Code** from the menu, and add the following references to the form.

```

using System.Reflection;
using System.Deployment.Application;
using Microsoft.Samples.ClickOnceOnDemand;
using System.Security.Permissions;

```

```

Imports System.Reflection
Imports System.Deployment.Application
Imports System.Collections.Generic
Imports Microsoft.Samples.ClickOnceOnDemand
Imports System.Security.Permissions

```

5. Add the following code to download this assembly on demand. This code shows how to map a set of assemblies to a group name using a generic [Dictionary](#) class. Because we are only downloading a single assembly in this walkthrough, there is only one assembly in our group. In a real application, you would likely want to download all assemblies related to a single feature in your application at the same time. The mapping table enables you to do this easily by associating all the DLLs that belong to a feature with a download group name.



```

// Maintain a dictionary mapping DLL names to download file groups. This is trivial for this sample,
// but will be important in real-world applications where a feature is spread across multiple DLLs,
// and you want to download all DLLs for that feature in one shot.
Dictionary<String, String> DllMapping = new Dictionary<String, String>();

[SecurityPermission(SecurityAction.Demand, ControlAppDomain=true)]
public Form1()
{
    InitializeComponent();

    DllMapping["ClickOnceLibrary"] = "ClickOnceLibrary";
    AppDomain.CurrentDomain.AssemblyResolve += new
    ResolveEventHandler(CurrentDomain_AssemblyResolve);
}

/*
 * Use ClickOnce APIs to download the assembly on demand.
 */
private Assembly CurrentDomain_AssemblyResolve(object sender, ResolveEventArgs args)
{
    Assembly newAssembly = null;

    if (ApplicationDeployment.IsNetworkDeployed)
    {
        ApplicationDeployment deploy = ApplicationDeployment.CurrentDeployment;

        // Get the DLL name from the Name argument.
        string[] nameParts = args.Name.Split(',');
        string dllName = nameParts[0];
        string downloadGroupName = DllMapping[dllName];

        try
        {
            deploy.DownloadFileGroup(downloadGroupName);
        }
        catch (DeploymentException de)
        {
            MessageBox.Show("Downloading file group failed. Group name: " + downloadGroupName + ";
DLL name: " + args.Name);
            throw (de);
        }

        // Load the assembly.
        // Assembly.Load() doesn't work here, as the previous failure to load the assembly
        // is cached by the CLR. LoadFrom() is not recommended. Use LoadFile() instead.
        try
        {
            newAssembly = Assembly.LoadFile(Application.StartupPath + @"\" + dllName + ".dll");
        }
        catch (Exception e)
        {
            throw (e);
        }
    }
    else
    {
        //Major error - not running under ClickOnce, but missing assembly. Don't know how to recover.
        throw (new Exception("Cannot load assemblies dynamically - application is not deployed using
ClickOnce."));
    }

    return (newAssembly);
}

```

```

' Maintain a dictionary mapping DLL names to download file groups. This is trivial for this sample,
' but will be important in real-world applications where a feature is spread across multiple DLLs,
' and you want to download all DLLs for that feature in one shot.
Dim DllMappingTable As New Dictionary(Of String, String)()

<SecurityPermission(SecurityAction.Demand, ControlAppDomain:=True)> _
Sub New()
    ' This call is required by the Windows Form Designer.
    InitializeComponent()

    ' Add any initialization after the InitializeComponent() call.
    DllMappingTable("ClickOnceLibrary") = "ClickOnceLibrary"
End Sub

Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
MyBase.Load
    AddHandler AppDomain.CurrentDomain.AssemblyResolve, AddressOf Me.CurrentDomain_AssemblyResolve
End Sub

Private Function CurrentDomain_AssemblyResolve(ByVal sender As Object, ByVal args As
ResolveEventArgs) As System.Reflection.Assembly
    Dim NewAssembly As Assembly = Nothing

    If (ApplicationDeployment.IsNetworkDeployed) Then
        Dim Deploy As ApplicationDeployment = ApplicationDeployment.CurrentDeployment

        ' Get the DLL name from the argument.
        Dim NameParts As String() = args.Name.Split(",")
        Dim DllName As String = NameParts(0)
        Dim DownloadGroupName As String = DllMappingTable(DllName)

        Try
            Deploy.DownloadFileGroup(DownloadGroupName)
        Catch ex As Exception
            MessageBox.Show("Could not download file group from Web server. Contact administrator.
Group name: " & DownloadGroupName & "; DLL name: " & args.Name)
            Throw (ex)
        End Try

        ' Load the assembly.
        ' Assembly.Load() doesn't work here, as the previous failure to load the assembly
        ' is cached by the CLR. LoadFrom() is not recommended. Use LoadFile() instead.
        Try
            NewAssembly = Assembly.LoadFile(Application.StartupPath & "\" & DllName & ".dll")
        Catch ex As Exception
            Throw (ex)
        End Try
    Else
        ' Major error - not running under ClickOnce, but missing assembly. Don't know how to recover.
        Throw New Exception("Cannot load assemblies dynamically - application is not deployed using
ClickOnce.")
    End If

    Return NewAssembly
End Function

```

- On the **View** menu, click **Toolbox**. Drag a **Button** from the **Toolbox** onto the form. Double-click the button and add the following code to the **Click** event handler.

```

private void getAssemblyButton_Click(object sender, EventArgs e)
{
    DynamicClass dc = new DynamicClass();
    MessageBox.Show("Message: " + dc.Message);
}

```

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
Button1.Click
    Dim DC As New DynamicClass()
    MessageBox.Show("Message is " & DC.Message)
End Sub
```

## Mark assemblies as optional

### To mark assemblies as optional in your ClickOnce application by using Visual Studio

1. Right-click the Windows Forms project in **Solution Explorer** and click **Properties**. Select the **Publish** tab.
2. Click the **Application Files** button.
3. Find the listing for *ClickOnceLibrary.dll*. Set the **Publish Status** drop-down box to **Include**.
4. Expand the **Group** drop-down box and select **New**. Enter the name `ClickOnceLibrary` as the new group name.
5. Continue publishing your application as described in [How to: Publish a ClickOnce application using the Publish Wizard](#).

### To mark assemblies as optional in your ClickOnce application by using Manifest Generation and Editing Tool — Graphical Client (MageUI.exe)

1. Create your ClickOnce manifests as described in [Walkthrough: Manually deploy a ClickOnce application](#).
2. Before closing MageUI.exe, select the tab that contains your deployment's application manifest, and within that tab select the **Files** tab.
3. Find ClickOnceLibrary.dll in the list of application files and set its **File Type** column to **None**. For the **Group** column, type `ClickOnceLibrary.dll`.

## Test the new assembly

To test your on-demand assembly:

1. Start your application deployed with ClickOnce.
2. When your main form appears, press the [Button](#). You should see a string in a message box window that reads, "Hello, World!"

## See also

- [ApplicationDeployment](#)

# Walkthrough: Download assemblies on demand with the ClickOnce deployment API

4/2/2021 • 5 minutes to read • [Edit Online](#)

By default, all of the assemblies included in a ClickOnce application are downloaded when the application is first run. However, you may have parts of your application that are used by a small set of your users. In this case, you want to download an assembly only when you create one of its types. The following walkthrough demonstrates how to mark certain assemblies in your application as "optional", and how to download them by using classes in the [System.Deployment.Application](#) namespace when the common language runtime (CLR) demands them.

## NOTE

Your application will have to run in full trust to use this procedure.

## Prerequisites

You will need one of the following components to complete this walkthrough:

- The Windows SDK. The Windows SDK can be downloaded from the Microsoft Download Center.
- Visual Studio.

## Create the projects

**To create a project that uses an on-demand assembly**

1. Create a directory named ClickOnceOnDemand.
2. Open the Windows SDK Command Prompt or the Visual Studio Command Prompt.
3. Change to the ClickOnceOnDemand directory.
4. Generate a public/private key pair using the following command:

```
sn -k TestKey.snk
```

5. Using Notepad or another text editor, define a class named `DynamicClass` with a single property named `Message`.

```
Public Class DynamicClass
    Sub New()

    End Sub

    Public ReadOnly Property Message() As String
        Get
            Message = "Hello, world!"
        End Get
    End Property
End Class
```

```

using System;
using System.Collections.Generic;
using System.Text;

namespace Microsoft.Samples.ClickOnceOnDemand
{
    public class DynamicClass
    {
        public DynamicClass() {}

        public string Message
        {
            get
            {
                return ("Hello, world!");
            }
        }
    }
}

```

6. Save the text as a file named *ClickOnceLibrary.cs* or *ClickOnceLibrary.vb*, depending on the language you use, to the *ClickOnceOnDemand* directory.
7. Compile the file into an assembly.

```
csc /target:library /keyfile:TestKey.snk ClickOnceLibrary.cs
```

```
vbc /target:library /keyfile:TestKey.snk ClickOnceLibrary.vb
```

8. To get the public key token for the assembly, use the following command:

```
sn -T ClickOnceLibrary.dll
```

9. Create a new file using your text editor and enter the following code. This code creates a Windows Forms application that downloads the ClickOnceLibrary assembly when it is required.

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Reflection;
using System.Deployment.Application;
using Microsoft.Samples.ClickOnceOnDemand;

namespace ClickOnceOnDemand
{
    [System.Security.Permissions.SecurityPermission(System.Security.Permissions.SecurityAction.Demand,
    Unrestricted=true)]
    public class Form1 : Form
    {
        // Maintain a dictionary mapping DLL names to download file groups. This is trivial for this
        sample,
        // but will be important in real-world applications where a feature is spread across multiple
        DLLs,
        // and you want to download all DLLs for that feature in one shot.
        Dictionary<String, String> DllMapping = new Dictionary<String, String>();
    }
}

```

```

public static void Main()
{
    Form1 NewForm = new Form1();
    Application.Run(NewForm);
}

public Form1()
{
    // Configure form.
    this.Size = new Size(500, 200);
    Button getAssemblyButton = new Button();
    getAssemblyButton.Size = new Size(130, getAssemblyButton.Size.Height);
    getAssemblyButton.Text = "Test Assembly";
    getAssemblyButton.Location = new Point(50, 50);
    this.Controls.Add(getAssemblyButton);
    getAssemblyButton.Click += new EventHandler(getAssemblyButton_Click);

    DllMapping["ClickOnceLibrary"] = "ClickOnceLibrary";
    AppDomain.CurrentDomain.AssemblyResolve += new
ResolveEventHandler(CurrentDomain_AssemblyResolve);
}

/*
 * Use ClickOnce APIs to download the assembly on demand.
 */
private Assembly CurrentDomain_AssemblyResolve(object sender, ResolveEventArgs args)
{
    Assembly newAssembly = null;

    if (ApplicationDeployment.IsNetworkDeployed)
    {
        ApplicationDeployment deploy = ApplicationDeployment.CurrentDeployment;

        // Get the DLL name from the Name argument.
        string[] nameParts = args.Name.Split(',');
        string dllName = nameParts[0];
        string downloadGroupName = DllMapping[dllName];

        try
        {
            deploy.DownloadFileGroup(downloadGroupName);
        }
        catch (DeploymentException de)
        {
            MessageBox.Show("Downloading file group failed. Group name: " + downloadGroupName
+ "; DLL name: " + args.Name);
            throw (de);
        }

        // Load the assembly.
        // Assembly.Load() doesn't work here, as the previous failure to load the assembly
        // is cached by the CLR. LoadFrom() is not recommended. Use LoadFile() instead.
        try
        {
            newAssembly = Assembly.LoadFile(Application.StartupPath + @"\" + dllName +
".dll," +
"Version=1.0.0.0, Culture=en, PublicKeyToken=03689116d3a4ae33");
        }
        catch (Exception e)
        {
            throw (e);
        }
    }
    else
    {
        //Major error - not running under ClickOnce, but missing assembly. Don't know how to
recover.

        throw (new Exception("Cannot load assemblies dynamically - application is not

```

```

        deployed using ClickOnce."));
    }

    return (newAssembly);
}

private void getAssemblyButton_Click(object sender, EventArgs e)
{
    DynamicClass dc = new DynamicClass();
    MessageBox.Show("Message: " + dc.Message);
}
}
}

```

```

Imports System
Imports System.Windows.Forms
Imports System.Deployment.Application
Imports System.Drawing
Imports System.Reflection
Imports System.Collections.Generic
Imports Microsoft.Samples.ClickOnceOnDemand

Namespace Microsoft.Samples.ClickOnceOnDemand
    <System.Security.Permissions.SecurityPermission(System.Security.Permissions.SecurityAction.Demand,
    Unrestricted:=true)> _
    Class Form1
        Inherits Form

        ' Maintain a dictionary mapping DLL names to download file groups. This is trivial for this
        sample,
        ' but will be important in real-world applications where a feature is spread across multiple
        DLLs,
        ' and you want to download all DLLs for that feature in one shot.
        Dim DllMapping as Dictionary(Of String, String) = new Dictionary(of String, String)()

        Public Sub New()
            ' Add button to form.
            Dim GetAssemblyButton As New Button()
            GetAssemblyButton.Location = New Point(100, 100)
            GetAssemblyButton.Text = "Get assembly on demand"
            AddHandler GetAssemblyButton.Click, AddressOf GetAssemblyButton_Click

            Me.Controls.Add(GetAssemblyButton)

            DllMapping("ClickOnceLibrary") = "ClickOnceLibrary"
            AddHandler AppDomain.CurrentDomain.AssemblyResolve, AddressOf
            CurrentDomain_AssemblyResolve
        End Sub

        <STAThread()> _
        Shared Sub Main()
            Application.EnableVisualStyles()
            Application.Run(New Form1())
        End Sub

        Private Function CurrentDomain_AssemblyResolve(ByVal sender As Object, ByVal args As
        ResolveEventArgs) As Assembly
            If ApplicationDeployment.IsNetworkDeployed Then
                Dim deploy As ApplicationDeployment = ApplicationDeployment.CurrentDeployment

                ' Get the DLL name from the Name argument.
                Dim nameParts() as String = args.Name.Split(",")
                Dim dllName as String = nameParts(0)
                Dim downloadGroupName as String = DllMapping(dllName)

```

```

        Try
            deploy.DownloadFileGroup(downloadGroupName)
        Catch ex As DeploymentException

        End Try

        ' Load the assembly.
        Dim newAssembly As Assembly = Nothing

        Try
            newAssembly = Assembly.LoadFile(Application.StartupPath & "\\\" & dllName &
            ".dll," & _
            "Version=1.0.0.0, Culture=en, PublicKeyToken=03689116d3a4ae33")
        Catch ex As Exception
            MessageBox.Show("Could not download assembly on demand.")
        End Try

        CurrentDomain_AssemblyResolve = newAssembly
    Else
        CurrentDomain_AssemblyResolve = Nothing
    End If
End Function

Private Sub GetAssemblyButton_Click(ByVal sender As Object, ByVal e As EventArgs)
    Dim ourClass As New DynamicClass()
    MessageBox.Show("DynamicClass string is: " + ourClass.Message)
End Sub
End Class
End Namespace

```

10. In the code, locate the call to [LoadFile](#).
11. Set `PublicKeyToken` to the value that you retrieved earlier.
12. Save the file as either *Form1.cs* or *Form1.vb*.
13. Compile it into an executable using the following command.

```
csc /target:exe /reference:ClickOnceLibrary.dll Form1.cs
```

```
vbc /target:exe /reference:ClickOnceLibrary.dll Form1.vb
```

## Mark assemblies as optional

To mark assemblies as optional in your ClickOnce application by using MageUI.exe

1. Using *MageUI.exe*, create an application manifest as described in [Walkthrough: Manually deploy a ClickOnce application](#). Use the following settings for the application manifest:
  - Name the application manifest `ClickOnceOnDemand`.
  - On the **Files** page, in the *ClickOnceLibrary.dll* row, set the **File Type** column to **None**.
  - On the **Files** page, in the *ClickOnceLibrary.dll* row, type `ClickOnceLibrary.dll` in the **Group** column.
2. Using *MageUI.exe*, create a deployment manifest as described in [Walkthrough: Manually deploy a ClickOnce application](#). Use the following settings for the deployment manifest:
  - Name the deployment manifest `ClickOnceOnDemand`.



# Testing the new assembly

## To test your on-demand assembly

1. Upload your ClickOnce deployment to a Web server.
2. Start your application deployed with ClickOnce from a Web browser by entering the URL to the deployment manifest. If you call your ClickOnce application `ClickOnceOnDemand`, and you upload it to the root directory of adatum.com, your URL would look like this:

```
http://www.adatum.com/ClickOnceOnDemand/ClickOnceOnDemand.application
```

3. When your main form appears, press the [Button](#). You should see a string in a message box window that reads "Hello, World!".

## See also

- [ApplicationDeployment](#)

# Walkthrough: Download satellite assemblies on demand with the ClickOnce deployment API using the Designer

4/2/2021 • 3 minutes to read • [Edit Online](#)

Windows Forms applications can be configured for multiple cultures through the use of satellite assemblies. A *satellite assembly* is an assembly that contains application resources for a culture other than the application's default culture.

As discussed in [Localizing ClickOnce Applications](#), you can include multiple satellite assemblies for multiple cultures within the same ClickOnce deployment. By default, ClickOnce will download all of the satellite assemblies in your deployment to the client machine, although a single client will probably require only one satellite assembly.

This walkthrough demonstrates how to mark your satellite assemblies as optional, and download only the assembly a client machine needs for its current culture settings.

## NOTE

For testing purposes, the following code examples programmatically set the culture to `ja-JP`. See the "Next Steps" section later in this topic for information on how to adjust this code for a production environment.

### To mark satellite assemblies as optional

1. Build your project. This will generate satellite assemblies for all of the cultures you are localizing to.
2. Right-click on your project name in Solution Explorer, and click **Properties**.
3. Click the **Publish** tab, and then click **Application Files**.
4. Select the **Show all files** check box to display satellite assemblies. By default, all satellite assemblies will be included in your deployment and will be visible in this dialog box.

A satellite assembly will have a name in the form `<isoCode>\ApplicationName.resources.dll`, where `<isoCode>` is a language identifier in RFC 1766 format.

5. Click **New** in the **Download Group** list for each language identifier. When prompted for a download group name, enter the language identifier. For example, for a Japanese satellite assembly, you would specify the download group name `ja-JP`.
6. Close the **Application Files** dialog box.

### To download satellite assemblies on demand in C#

1. Open the `Program.cs` file. If you do not see this file in Solution Explorer, select your project, and on the **Project** menu, click **Show All Files**.
2. Use the following code to download the appropriate satellite assembly and start your application.

```

using System;
using System.Collections.Generic;
using System.Windows.Forms;
using System.Threading;
using System.Globalization;
using System.Deployment.Application;
using System.Reflection;

namespace ClickOnce.SatelliteAssemblies
{
    static class Program
    {
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Thread.CurrentThread.CurrentUICulture = new CultureInfo("ja-JP");

            // Call this before initializing the main form, which will cause the resource manager
            // to look for the appropriate satellite assembly.
            GetSatelliteAssemblies(Thread.CurrentThread.CurrentCulture.ToString());

            Application.Run(new Form1());
        }

        static void GetSatelliteAssemblies(string groupName)
        {
            if (ApplicationDeployment.IsNetworkDeployed)
            {
                ApplicationDeployment deploy = ApplicationDeployment.CurrentDeployment;

                if (deploy.IsFirstRun)
                {
                    try
                    {
                        deploy.DownloadFileGroup(groupName);
                    }
                    catch (DeploymentException de)
                    {
                        // Log error. Do not report this error to the user, because a satellite
                        // assembly may not exist if the user's culture and the application's
                        // default culture match.
                    }
                }
            }
        }
    }
}

```

### To download satellite assemblies on demand in Visual Basic

1. In the **Properties** window for the application, click the **Application** tab.
2. At the bottom of the tab page, click **View Application Events**.
3. Add the following imports to the beginning of the *ApplicationEvents.VB* file.

```

Imports System.Deployment.Application
Imports System.Globalization
Imports System.Threading

```

4. Add the following code to the `MyApplication` class.

```

Private Sub MyApplication_Startup(ByVal sender As Object, ByVal e As
Microsoft.VisualBasic.ApplicationServices.StartupEventArgs) Handles Me.Startup
    Thread.CurrentThread.CurrentUICulture = New CultureInfo("ja-JP")
    GetSatelliteAssemblies(Thread.CurrentThread.CurrentUICulture.ToString())
End Sub

Private Shared Sub GetSatelliteAssemblies(ByVal groupName As String)
    If (ApplicationDeployment.IsNetworkDeployed) Then

        Dim deploy As ApplicationDeployment = ApplicationDeployment.CurrentDeployment

        If (deploy.IsFirstRun) Then
            Try
                deploy.DownloadFileGroup(groupName)
            Catch de As DeploymentException
                ' Log error. Do not report this error to the user, because a satellite
                ' assembly may not exist if the user's culture and the application's
                ' default culture match.
            End Try
        End If
    End If
End Sub

```

## Next steps

In a production environment, you will likely need to remove the line in the code examples that sets [CurrentUICulture](#) to a specific value, because client machines will have the correct value set by default. When your application runs on a Japanese client machine, for example, [CurrentUICulture](#) will be `ja-JP` by default. Setting it programmatically is a good way to test your satellite assemblies before you deploy your application.

## See also

- [Walkthrough: Download satellite assemblies on demand with the ClickOnce deployment API](#)
- [Localize ClickOnce applications](#)

# Walkthrough: Manually deploy a ClickOnce application

3/5/2021 • 9 minutes to read • [Edit Online](#)

If you cannot use Visual Studio to deploy your ClickOnce application, or you need to use advanced deployment features, such as Trusted Application Deployment, you should use the *Mage.exe* command-line tool to create your ClickOnce manifests. This walkthrough describes how to create a ClickOnce deployment by using either the command-line version (*Mage.exe*) or the graphical version (*MageUI.exe*) of the Manifest Generation and Editing Tool.

## Prerequisites

This walkthrough has some prerequisites and options that you need to choose before building a deployment.

- Install *Mage.exe* and *MageUI.exe*.

*Mage.exe* and *MageUI.exe* are part of the Windows Software Development Kit (SDK). You must either have the Windows SDK installed or the version of the Windows SDK included with Visual Studio. For more information, see [Windows SDK](#) on MSDN.

- Provide an application to deploy.

This walkthrough assumes that you have a Windows application that you are ready to deploy. This application will be referred to as AppToDeploy.

- Determine how the deployment will be distributed.

The distribution options include: Web, file share, or CD. For more information, see [ClickOnce Security and Deployment](#).

- Determine whether the application requires an elevated level of trust.

If your application requires Full Trust—for example, full access to the user's system—you can use the `-TrustLevel` option of *Mage.exe* to set this. If you want to define a custom permission set for your application, you can copy the Internet or intranet permission section from another manifest, modify it to suit your needs, and add it to the application manifest using either a text editor or *MageUI.exe*. For more information, see [Trusted Application Deployment overview](#).

- Obtain an Authenticode certificate.

You should sign your deployment with an Authenticode certificate. You can generate a test certificate by using Visual Studio, *MageUI.exe*, or *MakeCert.exe* and *Pvk2Pfx.exe* tools, or you can obtain a certificate from a Certificate Authority (CA). If you choose to use Trusted Application Deployment, you must also perform a one-time installation of the certificate onto all client computers. For more information, see [Trusted Application Deployment Overview](#).

### NOTE

You can also sign your deployment with a CNG certificate that you can obtain from a Certificate Authority.

- Make sure that the application does not have a manifest with UAC information.

You need to determine whether your application contains a manifest with User Account Control (UAC) information, such as an `<dependentAssembly>` element. To examine an application manifest, you can use the Windows Sysinternals [Sigcheck](#) utility.

If your application contains a manifest with UAC details, you must re-build it without the UAC information. For a C# project in Visual Studio, open the project properties and select the Application tab. In the **Manifest** drop-down list, select **Create application without a manifest**. For a Visual Basic project in Visual Studio, open the project properties, select the Application tab, and click **View UAC Settings**. In the opened manifest file, remove all elements within the single `<asmv1:assembly>` element.

- Determine whether the application requires prerequisites on the client computer.

ClickOnce applications deployed from Visual Studio can include a prerequisite installation bootstrapper (*setup.exe*) with your deployment. This walkthrough creates the two manifests required for a ClickOnce deployment. You can create a prerequisite bootstrapper by using the [GenerateBootstrapper](#) task.

### To deploy an application with the Mage.exe command-line tool

1. Create a directory where you will store your ClickOnce deployment files.
2. In the deployment directory you just created, create a version subdirectory. If this is the first time that you are deploying the application, name the version subdirectory **1.0.0.0**.

#### NOTE

The version of your deployment can be distinct from the version of your application.

3. Copy all of your application files to the version subdirectory, including executable files, assemblies, resources, and data files. If necessary, you can create additional subdirectories that contain additional files.
4. Open the Windows SDK or Visual Studio command prompt and change to the version subdirectory.
5. Create the application manifest with a call to *Mage.exe*. The following statement creates an application manifest for code compiled to run on the Intel x86 processor.

```
mage -New Application -Processor x86 -ToFile AppToDeploy.exe.manifest -name "My App" -Version 1.0.0.0 -FromDirectory .
```

#### NOTE

Be sure to include the dot (.) after the `-FromDirectory` option, which indicates the current directory. If you do not include the dot, you must specify the path to your application files.

6. Sign the application manifest with your Authenticode certificate. Replace *mycert.pfx* with the path to your certificate file. Replace *passwd* with the password for your certificate file.

```
mage -Sign AppToDeploy.exe.manifest -CertFile mycert.pfx -Password passwd
```

Starting with the .NET Framework 4.6.2 SDK, which is distributed with Visual Studio and with the Windows SDK, *mage.exe* signs manifests with CNG as well as with Authenticode certificates. Use the same command line parameters as with Authenticode certificates.

7. Change to the root of the deployment directory.

8. Generate the deployment manifest with a call to *Mage.exe*. By default, *Mage.exe* will mark your ClickOnce deployment as an installed application, so that it can be run both online and offline. To make the application available only when the user is online, use the `-Install` option with a value of `false`. If you use the default, and users will install your application from a Web site or file share, make sure that the value of the `-ProviderUrl` option points to the location of the application manifest on the Web server or share.

```
mage -New Deployment -Processor x86 -Install true -Publisher "My Co." -ProviderUrl  
"\\myServer\myShare\AppToDeploy.application" -AppManifest 1.0.0.0\AppToDeploy.exe.manifest -ToFile  
AppToDeploy.application
```

9. Sign the deployment manifest with your Authenticode or CNG certificate.

```
mage -Sign AppToDeploy.application -CertFile mycert.pfx -Password passwd
```

10. Copy all of the files in the deployment directory to the deployment destination or media. This may be either a folder on a Web site or FTP site, a file share, or a CD-ROM.
11. Provide your users with the URL, UNC, or physical media required to install your application. If you provide a URL or a UNC, you must give your users the full path to the deployment manifest. For example, if AppToDeploy is deployed to `http://webserver01/` in the AppToDeploy directory, the full URL path would be `http://webserver01/AppToDeploy/AppToDeploy.application`.

### To deploy an application with the MageUI.exe graphical tool

1. Create a directory where you will store your ClickOnce deployment files.
2. In the deployment directory you just created, create a version subdirectory. If this is the first time that you are deploying the application, name the version subdirectory `1.0.0.0`.

#### NOTE

The version of your deployment is probably distinct from the version of your application.

3. Copy all of your application files to the version subdirectory, including executable files, assemblies, resources, and data files. If necessary, you can create additional subdirectories that contain additional files.
4. Start the *MageUI.exe* graphical tool.

MageUI.exe

5. Create a new application manifest by selecting **File, New, Application Manifest** from the menu.
6. On the default **Name** tab, type the name and version number of this deployment. Also specify the **Processor** that your application is built for, such as x86.
7. Select the **Files** tab and then select the ellipsis (...) button next to the **Application directory** text box. A **Browse For Folder** dialog box appears.
8. Select the version subdirectory containing your application files, and then select **OK**.
9. If you will deploy from Internet Information Services (IIS), select the **When populating add the .deploy extension to any file that does not have it** check box.
10. Go to the **Populate** button to add all your application files to the file list. If your application contains

more than one executable file, mark the main executable file for this deployment as the startup application by selecting **Entry Point** from the **File Type** drop-down list. (If your application contains only one executable file, *MageUI.exe* will mark it for you.)

11. Select the **Permissions Required** tab and select the level of trust that you need your application to assert. The default is **FullTrust**, which will be suitable for most applications.
12. Select **File, Save As** from the menu. A Signing Options dialog box appears prompting you to sign the application manifest.
13. If you have a certificate stored as a file on your file system, use the **Sign with certificate file** option, and select the certificate from the file system by using the ellipsis (...) button. Then type your certificate password.

-or-

If your certificate is kept in a certificate store accessible from your computer, select the **Sign with stored certificate** option, and select the certificate from the provided list.

14. Select **OK** to sign your application manifest. The **Save As** dialog box appears.
15. In the **Save As** dialog box, specify the version directory, and then select **Save**.
16. Select **File, New, Deployment Manifest** from the menu to create your deployment manifest.
17. On the **Name** tab, specify a name and version number for this deployment (**1.0.0.0** in this example). Also specify the **Processor** that your application is built for, such as **x86**.
18. Select the **Description** tab, and specify values for **Publisher** and **Product**. (**Product** is the name given to your application on the Windows Start menu when your application installs on a client computer for offline use.)
19. Select the **Deployment Options** tab, and in the **Start Location** text box, specify the location of the application manifest on the Web server or share. For example,  
*\\myServer\myShare\AppToDeploy.application*.
20. If you added the *.deploy* extension in a previous step, also select **Use .deploy file name extension** here.
21. Select the **Update Options** tab, and specify how often you would like this application to update. If your application uses [UpdateCheckInfo](#) to check for updates itself, clear the **This application should check for updates** check box.
22. Select the **Application Reference** tab and then go to the **Select Manifest** button. An open dialog box appears.
23. Select the application manifest that you created earlier and then select **Open**.
24. Select **File, Save As** from the menu. A **Signing Options** dialog box appears prompting you to sign the deployment manifest.
25. If you have a certificate stored as a file on your file system, use the **Sign with certificate file** option, and select the certificate from the file system by using the ellipsis (...) button. Then type your certificate password.

-or-

If your certificate is kept in a certificate store accessible from your computer, select the **Sign with stored certificate** option, and select the certificate from the provided list.

26. Go to **OK** to sign your deployment manifest. The **Save As** dialog box appears.



27. In the **Save As** dialog box, move up one directory to the root of your deployment and then select **Save**.
28. Copy all of the files in the deployment directory to the deployment destination or media. This may be either a folder on a Web site or FTP site, a file share, or a CD-ROM.
29. Provide your users with the URL, UNC, or physical media required to install your application. If you provide a URL or a UNC, you must give your users the full path the deployment manifest. For example, if AppToDeploy is deployed to http://webserver01/ in the AppToDeploy directory, the full URL path would be http://webserver01/AppToDeploy/AppToDeploy.application.

## Next steps

When you need to deploy a new version of the application, create a new directory named after the new version—for example, 1.0.0.1—and copy the new application files into the new directory. Next, you need to follow the previous steps to create and sign a new application manifest, and update and sign the deployment manifest. Be careful to specify the same higher version in both the *Mage.exe* `-New` and `-Update` calls, as ClickOnce only updates higher versions, with the left-most integer most significant. If you used *MageUI.exe*, you can update the deployment manifest by opening it, selecting the **Application Reference** tab, go to the **Select Manifest** button, and then selecting the updated application manifest.

## See also

- [Mage.exe \(Manifest Generation and Editing Tool\)](#)
- [MageUI.exe \(Manifest Generation and Editing Tool, Graphical Client\)](#)
- [Publish ClickOnce applications](#)
- [ClickOnce deployment manifest](#)
- [ClickOnce application manifest](#)

# Walkthrough: Manually deploy a ClickOnce application that does not require re-signing and that preserves branding information

3/5/2021 • 6 minutes to read • [Edit Online](#)

When you create a ClickOnce application and then give it to a customer to publish and deploy, the customer has traditionally had to update the deployment manifest and re-sign it. While that is still the preferred method in most cases, the .NET Framework 3.5 enables you to create ClickOnce deployments that can be deployed by customers without having to regenerate a new deployment manifest. For more information, see [Deploy ClickOnce applications for testing and production servers without resigning](#).

When you create a ClickOnce application and then give it to a customer to publish and deploy, the application can use the customer's branding or can preserve your branding. For example, if the application is a single proprietary application, you might want to preserve your branding. If the application is highly customized for each customer, you might want to use the customer's branding. The .NET Framework 3.5 enables you to preserve your branding, publisher information and security signature when you give an application to an organization to deploy. For more information, see [Create ClickOnce applications for others to deploy](#).

## NOTE

In this walkthrough you create deployments manually by using either the command-line tool *Mage.exe* or the graphical tool *MageUI.exe*. For more information about manual deployments, see [Walkthrough: Manually deploy a ClickOnce application](#).

## Prerequisites

To perform the steps in this walkthrough you need the following:

- A Windows Forms application that you are ready to deploy. This application will be referred to as *WindowsFormsApp1*.
- Visual Studio or the Windows SDK.

### To deploy a ClickOnce application with multiple deployment and branding support using *Mage.exe*

1. Open a Visual Studio command prompt or a Windows SDK command prompt, and change to the directory in which you will store your ClickOnce files.
2. Create a directory named after the current version of your deployment. If this is the first time that you are deploying the application, you will likely choose **1.0.0.0**.

## NOTE

The version of your deployment may be distinct from the version of your application files.

3. Create a subdirectory named **bin** and copy all of your application files here, including executable files, assemblies, resources, and data files.
4. Generate the application manifest with a call to *Mage.exe*.

```
mage -New Application -ToFile 1.0.0.0\WindowsFormsApp1.exe.manifest -Name "Windows Forms App 1" -
Version 1.0.0.0 -FromDirectory 1.0.0.0\bin -UseManifestForTrust true -Publisher "A. Datum
Corporation"
```

5. Sign the application manifest with your digital certificate.

```
mage -Sign WindowsFormsApp1.exe.manifest -CertFile mycert.pfx
```

6. Generate the deployment manifest with a call to *Mage.exe*. By default, *Mage.exe* will mark your ClickOnce deployment as an installed application, so that it can be run both online and offline. To make the application available only when the user is online, use the `-i` argument with a value of `f`. Since this application will take advantage of the multiple deployment feature, exclude the `-providerUrl` argument to *Mage.exe*. (In versions of the .NET Framework prior to version 3.5, excluding `-providerUrl` for an offline application will result in an error.)

```
mage -New Deployment -ToFile WindowsFormsApp1.application -Name "Windows Forms App 1" -Version
1.0.0.0 -AppManifest 1.0.0.0\WindowsFormsApp1.manifest
```

7. Do not sign the deployment manifest.
8. Provide all of the files to the customer, who will deploy the application on his network.
9. At this point, the customer must sign the deployment manifest with his own self-generated certificate. For example, if the customer works for a company named Adventure Works, he can generate a self-signed certificate using the *MakeCert.exe* tool. Next, use the *Pvk2pfx.exe* tool to combine the files created by *MakeCert.exe* into a PFX file that can be passed to *Mage.exe*.

```
makecert -r -pe -n "CN=Adventure Works" -sv MyCert.pvk MyCert.cer
pvk2pfx.exe -pvk MyCert.pvk -spc MyCert.cer -pfx MyCert.pfx
```

10. The customer next uses this certificate to sign the deployment manifest.

```
mage -Sign WindowsFormsApp1.application -CertFile MyCert.pfx
```

11. The customer deploys the application to their users.

### To deploy a ClickOnce application with multiple deployment and branding support using *MageUI.exe*

1. Open a Visual Studio command prompt or a Windows SDK command prompt, and navigate to the directory in which you will store your ClickOnce files.
2. Create a subdirectory named **bin** and copy all of your application files here, including executable files, assemblies, resources, and data files.
3. Create a subdirectory named after the current version of your deployment. If this is the first time that you are deploying the application, you will likely choose **1.0.0.0**.

#### NOTE

The version of your deployment may be distinct from the version of your application files.

4. Move the **bin** directory into the directory you created in step 2.
5. Start the graphical tool *MageUI.exe*.

MageUI.exe

6. Create a new application manifest by selecting **File, New, Application Manifest** from the menu.
7. On the default **Name** tab, enter the name and version number of this deployment. Also, supply a value for **Publisher**, which will be used as the folder name for the application's shortcut link in the Start menu when it is deployed.
8. Select the **Application Options** tab and click **Use Application Manifest for Trust Information**. This will enable third-party branding for this ClickOnce application.
9. Select the **Files** tab and click the **Browse** button next to the **Application Directory** text box.
10. Select the directory that contains your application files that you created in step 2, and click **OK** on the folder selection dialog box.
11. Click the **Populate** button to add all your application files to the file list. If your application contains more than one executable file, mark the main executable file for this deployment as the startup application by selecting **Entry Point** from the **File Type** drop-down list. (If your application only contains one executable file, *MageUI.exe* will mark it for you.)
12. Select the **Permissions Required** tab and select the level of trust you need your application to assert. The default is **Full Trust**, which will be appropriate for most applications.
13. Select **File, Save** from the menu, and save the application manifest. You will be prompted to sign the application manifest when you save it.
14. If you have a certificate stored as a file on your file system, use the **Sign as certificate file** option, and select the certificate from the file system using the ellipsis (...) button.

-or-

If your certificate is kept in a certificate store that can be accessed from your computer, select the **Sign with stored certificate option**, and select the certificate from the list provided.

15. Select **File, New, Deployment Manifest** from the menu to create your deployment manifest, and then on the **Name** tab, supply a name and version number (**1.0.0.0** in this example).
16. Switch to the **Update** tab, and specify how often you want this application to update. If your application uses the ClickOnce Deployment API to check for updates itself, clear the check box labeled **This application should check for updates**.
17. Switch to the **Application Reference** tab. You can pre-populate all of the values on this tab by clicking the **Select Manifest** button and selecting the application manifest you created in previous steps.
18. Choose **Save** and save the deployment manifest to disk. You will be prompted to sign the application manifest when you save it. Click **Cancel** to save the manifest without signing it.
19. Provide all of the application files to the customer.
20. At this point, the customer must sign the deployment manifest with his own self-generated certificate. For example, if the customer works for a company named Adventure Works, he can generate a self-signed certificate using the *MakeCert.exe* tool. Next, use the *Pvk2pfx.exe* tool to combine the files created by *MakeCert.exe* into a PFX file that can be passed to *MageUI.exe*.

```
makecert -r -pe -n "CN=Adventure Works" -sv MyCert.pvk MyCert.cer  
pvk2pfx.exe -pvk MyCert.pvk -spc MyCert.cer -pfx MyCert.pfx
```

21. With the certificate generated, the customer now signs the deployment manifest by opening the deployment manifest in *MageUI.exe*, and then saving it. When the signing dialog box appears, the customer selects **Sign as certificate file** option, and chooses the PFX file he has saved on disk.
22. The customer deploys the application to their users.

## See also

- [Mage.exe \(Manifest Generation and Editing Tool\)](#)
- [MageUI.exe \(Manifest Generation and Editing Tool, Graphical Client\)](#)
- [MakeCert](#)

# Walkthrough: Download satellite assemblies on demand with the ClickOnce deployment API

4/2/2021 • 3 minutes to read • [Edit Online](#)

Windows Forms applications can be configured for multiple cultures through the use of satellite assemblies. A *satellite assembly* is an assembly that contains application resources for a culture other than the application's default culture.

As discussed in [Localize ClickOnce applications](#), you can include multiple satellite assemblies for multiple cultures within the same ClickOnce deployment. By default, ClickOnce will download all of the satellite assemblies in your deployment to the client machine, although a single client will probably require only one satellite assembly.

This walkthrough demonstrates how to mark your satellite assemblies as optional, and download only the assembly a client machine needs for its current culture settings. The following procedure uses the tools available in the Windows Software Development Kit (SDK). You can also perform this task in Visual Studio. Also see [Walkthrough: Download satellite assemblies on demand with the ClickOnce deployment API using the Designer](#) or [Walkthrough: Download satellite assemblies on demand with the ClickOnce deployment API using the Designer](#).

## NOTE

For testing purposes, the following code example programmatically sets the culture to `ja-JP`. See the "Next Steps" section later in this topic for information on how to adjust this code for a production environment.

## Prerequisites

This topic assumes that you know how to add localized resources to your application using Visual Studio. For detailed instructions, see [Walkthrough: Localize Windows forms](#).

### To download satellite assemblies on demand

1. Add the following code to your application to enable on-demand downloading of satellite assemblies.

```

using System;
using System.Collections.Generic;
using System.Windows.Forms;
using System.Threading;
using System.Globalization;
using System.Deployment.Application;
using System.Reflection;

namespace ClickOnce.SatelliteAssemblies
{
    static class Program
    {
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Thread.CurrentThread.CurrentUICulture = new CultureInfo("ja-JP");

            // Call this before initializing the main form, which will cause the resource manager
            // to look for the appropriate satellite assembly.
            GetSatelliteAssemblies(Thread.CurrentThread.CurrentCulture.ToString());

            Application.Run(new Form1());
        }

        static void GetSatelliteAssemblies(string groupName)
        {
            if (ApplicationDeployment.IsNetworkDeployed)
            {
                ApplicationDeployment deploy = ApplicationDeployment.CurrentDeployment;

                if (deploy.IsFirstRun)
                {
                    try
                    {
                        deploy.DownloadFileGroup(groupName);
                    }
                    catch (DeploymentException de)
                    {
                        // Log error. Do not report error to the user, as there may not be a
                        // assembly if the user's culture and the application's default culture
                        match.
                    }
                }
            }
        }
    }
}

```

```
Imports System.Deployment.Application
Imports System.Globalization
Imports System.Threading

Public Class Form1
    Shared Sub Main(ByVal args As String())
        Application.EnableVisualStyles()

        Thread.CurrentThread.CurrentUICulture = New CultureInfo("ja-JP")
        GetSatelliteAssemblies(Thread.CurrentThread.CurrentUICulture.ToString())

        Application.Run(New Form1())
    End Sub

    Private Shared Sub GetSatelliteAssemblies(ByVal groupName As String)
        If (ApplicationDeployment.IsNetworkDeployed) Then

            Dim deploy As ApplicationDeployment = ApplicationDeployment.CurrentDeployment

            If (deploy.IsFirstRun) Then
                Try
                    deploy.DownloadFileGroup(groupName)
                Catch de As DeploymentException
                    ' Log error. Do not report error to the user, as there may not be a satellite
                    ' assembly if the user's culture and the application's default culture match.

                End Try
            End If
        End If
    End Sub
End Class
```

2. Generate satellite assemblies for your application by using [Resgen.exe \(Resource File Generator\)](#) or Visual Studio.
3. Generate an application manifest, or open your existing application manifest, by using *MageUI.exe*. For more information about this tool, see [MageUI.exe \(Manifest Generation and Editing Tool, Graphical Client\)](#).
4. Click the **Files** tab.
5. Click the **ellipsis** button (...) and select the directory containing all of your application's assemblies and files, including the satellite assemblies you generated using *Resgen.exe*. (A satellite assembly will have a name in the form *<isoCode>\ApplicationName.resources.dll*, where *<isoCode>* is a language identifier in RFC 1766 format.)
6. Click **Populate** to add the files to your deployment.
7. Select the **Optional** check box for each satellite assembly.
8. Set the group field for each satellite assembly to its ISO language identifier. For example, for a Japanese satellite assembly, you would specify a download group name of `ja-JP`. This will enable the code you added in step 1 to download the appropriate satellite assembly, depending upon the user's [CurrentUICulture](#) property setting.

## Next steps

In a production environment, you will likely need to remove the line in the code example that sets [CurrentUICulture](#) to a specific value, because client machines will have the correct value set by default. When your application runs on a Japanese client machine, for example, [CurrentUICulture](#) will be `ja-JP` by default. Setting this value programmatically is a good way to test your satellite assemblies before you deploy your



application.

## See also

- [Localize ClickOnce applications](#)

# Walkthrough: Create a custom installer for a ClickOnce application

4/2/2021 • 6 minutes to read • [Edit Online](#)

Any ClickOnce application based on an .exe file can be silently installed and updated by a custom installer. A custom installer can implement custom user experience during installation, including custom dialog boxes for security and maintenance operations. To perform installation operations, the custom installer uses the [InPlaceHostingManager](#) class. This walkthrough demonstrates how to create a custom installer that silently installs a ClickOnce application.

## Prerequisites

### To create a custom ClickOnce application installer

1. In your ClickOnce application, add references to System.Deployment and System.Windows.Forms.
2. Add a new class to your application and specify any name. This walkthrough uses the name `MyInstaller`.
3. Add the following `Imports` or `using` directives to the top of your new class.

```
Imports System.Deployment.Application
Imports System.Windows.Forms
```

```
using System.Deployment.Application;
using System.Windows.Forms;
```

4. Add the following methods to your class.

These methods call [InPlaceHostingManager](#) methods to download the deployment manifest, assert appropriate permissions, ask the user for permission to install, and then download and install the application into the ClickOnce cache. A custom installer can specify that a ClickOnce application is pre-trusted, or can defer the trust decision to the [AssertApplicationRequirements](#) method call. This code pre-trusts the application.

#### NOTE

Permissions assigned by pre-trusting cannot exceed the permissions of the custom installer code.

```
Dim WithEvents iphm As InPlaceHostingManager = Nothing

Public Sub InstallApplication(ByVal deployManifestUriStr As String)
    Try
        Dim deploymentUri As New Uri(deployManifestUriStr)
        iphm = New InPlaceHostingManager(deploymentUri, False)
        MessageBox.Show("Created the object.")
    Catch uriEx As UriFormatException
        MessageBox.Show("Cannot install the application: " & _
            "The deployment manifest URL supplied is not a valid URL." & _
            "Error: " & uriEx.Message)
    Return
    Catch platformEx As PlatformNotSupportedException
        MessageBox.Show("Cannot install the application: " &
```

```

        "This program requires Windows XP or higher. " & _
        "Error: " & platformEx.Message)

    Return
Catch argumentEx As ArgumentException
    MessageBox.Show("Cannot install the application: " & _
        "The deployment manifest URL supplied is not a valid URL." & _
        "Error: " & argumentEx.Message)

    Return
End Try

iphm.GetManifestAsync()
End Sub

Private Sub iphm_GetManifestCompleted(ByVal sender As Object, ByVal e As
GetManifestCompletedEventArgs) Handles iphm.GetManifestCompleted
    ' Check for an error.
    If (e.Error IsNot Nothing) Then
        ' Cancel download and install.
        MessageBox.Show("Could not download manifest. Error: " & e.Error.Message)
        Return
    End If

    ' Dim isFullTrust As Boolean = CheckForFullTrust(e.ApplicationManifest)

    ' Verify this application can be installed.
    Try
        ' the true parameter allows InPlaceHostingManager
        ' to grant the permissions requested in the application manifest.
        iphm.AssertApplicationRequirements(True)
    Catch ex As Exception
        MessageBox.Show("An error occurred while verifying the application. " & _
            "Error text: " & ex.Message)

        Return
    End Try

    ' Use the information from GetManifestCompleted() to confirm
    ' that the user wants to proceed.
    Dim appInfo As String = "Application Name: " & e.ProductName
    appInfo &= ControlChars.Lf & "Version: " & e.Version.ToString()
    appInfo &= ControlChars.Lf & "Support/Help Requests: "

    If Not (e.SupportUri Is Nothing) Then
        appInfo &= e.SupportUri.ToString()
    Else
        appInfo &= "N/A"
    End If

    appInfo &= ControlChars.Lf & ControlChars.Lf & _
        "Confirmed that this application can run with its requested permissions."

    ' If isFullTrust Then
    '     appInfo &= ControlChars.Lf & ControlChars.Lf & _
    '         "This application requires full trust in order to run."
    ' End If

    appInfo &= ControlChars.Lf & ControlChars.Lf & "Proceed with installation?"

    Dim dr As DialogResult = MessageBox.Show(appInfo, _
        "Confirm Application Install", MessageBoxButtons.OKCancel, MessageBoxIcon.Question)
    If dr <> System.Windows.Forms.DialogResult.OK Then
        Return
    End If

    ' Download the deployment manifest.
    ' Usually, this shouldn't throw an exception unless
    ' AssertApplicationRequirements() failed, or you did not call that method
    ' before calling this one.
    Try
        iphm.DownloadApplicationAsync()

```

```

        iphm.DownloadApplicationAsync()
    Catch downloadEx As Exception
        MessageBox.Show("Cannot initiate download of application. Error: " & downloadEx.Message)
    Return
    End Try
End Sub

#If 0 Then
    Private Function CheckForFullTrust(ByVal appManifest As XmlReader) As Boolean
        Dim isFullTrust As Boolean = False

        If (appManifest Is Nothing) Then
            Throw New ArgumentNullException("appManifest cannot be null.")
        End If

        Dim xaUnrestricted As XAttribute
        xaUnrestricted = XDocument.Load(appManifest) _
            .Element("{urn:schemas-microsoft-com:asm.v1}assembly") _
            .Element("{urn:schemas-microsoft-com:asm.v2}trustInfo") _
            .Element("{urn:schemas-microsoft-com:asm.v2}security") _
            .Element("{urn:schemas-microsoft-com:asm.v2}applicationRequestMinimum") _
            .Element("{urn:schemas-microsoft-com:asm.v2}PermissionSet") _
            .Attribute("Unrestricted") ' Attributes never have a namespace

        If xaUnrestricted Then
            If xaUnrestricted.Value = "true" Then
                Return True
            End If
        End If

        Return False
    End Function
#End If

    Private Sub iphm_DownloadProgressChanged(ByVal sender As Object, ByVal e As
DownloadProgressChangedEventArgs) Handles iphm.DownloadProgressChanged
        ' you can show percentage of task completed using e.ProgressPercentage
    End Sub

    Private Sub iphm_DownloadApplicationCompleted(ByVal sender As Object, ByVal e As
DownloadApplicationCompletedEventArgs) Handles iphm.DownloadApplicationCompleted
        ' Check for an error.
        If (e.Error IsNot Nothing) Then
            ' Cancel download and install.
            MessageBox.Show("Could not download and install application. Error: " & e.Error.Message)
            Return
        End If

        ' Inform the user that their application is ready for use.
        MessageBox.Show("Application installed! You may now run it from the Start menu.")
    End Sub

```

```

InPlaceHostingManager iphm = null;

public void InstallApplication(string deployManifestUriStr)
{
    try
    {
        Uri deploymentUri = new Uri(deployManifestUriStr);
        iphm = new InPlaceHostingManager(deploymentUri, false);
    }
    catch (UriFormatException uriEx)
    {
        MessageBox.Show("Cannot install the application: " +
            "The deployment manifest URL supplied is not a valid URL. " +
            "Error: " + uriEx.Message);
    }
    return;
}

```

```

    }
    catch (PlatformNotSupportedException platformEx)
    {
        MessageBox.Show("Cannot install the application: " +
            "This program requires Windows XP or higher. " +
            "Error: " + platformEx.Message);
        return;
    }
    catch (ArgumentException argumentEx)
    {
        MessageBox.Show("Cannot install the application: " +
            "The deployment manifest URL supplied is not a valid URL. " +
            "Error: " + argumentEx.Message);
        return;
    }

    iphm.GetManifestCompleted += new EventHandler<GetManifestCompletedEventArgs>
(iphm_GetManifestCompleted);
    iphm.GetManifestAsync();
}

void iphm_GetManifestCompleted(object sender, GetManifestCompletedEventArgs e)
{
    // Check for an error.
    if (e.Error != null)
    {
        // Cancel download and install.
        MessageBox.Show("Could not download manifest. Error: " + e.Error.Message);
        return;
    }

    // bool isFullTrust = CheckForFullTrust(e.ApplicationManifest);

    // Verify this application can be installed.
    try
    {
        // the true parameter allows InPlaceHostingManager
        // to grant the permissions requested in the applicaiton manifest.
        iphm.AssertApplicationRequirements(true) ;
    }
    catch (Exception ex)
    {
        MessageBox.Show("An error occurred while verifying the application. " +
            "Error: " + ex.Message);
        return;
    }

    // Use the information from GetManifestCompleted() to confirm
    // that the user wants to proceed.
    string appInfo = "Application Name: " + e.ProductName;
    appInfo += "\nVersion: " + e.Version;
    appInfo += "\nSupport/Help Requests: " + (e.SupportUri != null ?
        e.SupportUri.ToString() : "N/A");
    appInfo += "\n\nConfirmed that this application can run with its requested permissions.";
    // if (isFullTrust)
    // appInfo += "\n\nThis application requires full trust in order to run.";
    appInfo += "\n\nProceed with installation?";

    DialogResult dr = MessageBox.Show(appInfo, "Confirm Application Install",
        MessageBoxButtons.OKCancel, MessageBoxIcon.Question);
    if (dr != System.Windows.Forms.DialogResult.OK)
    {
        return;
    }

    // Download the deployment manifest.
    iphm.DownloadProgressChanged += new EventHandler<DownloadProgressChangedEventArgs>
(iphm_DownloadProgressChanged);
    iphm.DownloadApplicationCompleted += new EventHandler<DownloadApplicationCompletedEventArgs>

```

```

iphm.DownloadApplicationCompleted += new EventHandler<DownloadApplicationCompletedEventArgs>
(iphm_DownloadApplicationCompleted);

try
{
    // Usually this shouldn't throw an exception unless AssertApplicationRequirements() failed,
    // or you did not call that method before calling this one.
    iphm.DownloadApplicationAsync();
}
catch (Exception downloadEx)
{
    MessageBox.Show("Cannot initiate download of application. Error: " +
        downloadEx.Message);
    return;
}
}

/*
private bool CheckForFullTrust(XmlReader appManifest)
{
    if (appManifest == null)
    {
        throw (new ArgumentNullException("appManifest cannot be null."));
    }

    XAttribute xaUnrestricted =
        XDocument.Load(appManifest)
            .Element("{urn:schemas-microsoft-com:asm.v1}assembly")
            .Element("{urn:schemas-microsoft-com:asm.v2}trustInfo")
            .Element("{urn:schemas-microsoft-com:asm.v2}security")
            .Element("{urn:schemas-microsoft-com:asm.v2}applicationRequestMinimum")
            .Element("{urn:schemas-microsoft-com:asm.v2}PermissionSet")
            .Attribute("Unrestricted"); // Attributes never have a namespace

    if (xaUnrestricted != null)
        if (xaUnrestricted.Value == "true")
            return true;

    return false;
}
*/

void iphm_DownloadApplicationCompleted(object sender, DownloadApplicationCompletedEventArgs e)
{
    // Check for an error.
    if (e.Error != null)
    {
        // Cancel download and install.
        MessageBox.Show("Could not download and install application. Error: " + e.Error.Message);
        return;
    }

    // Inform the user that their application is ready for use.
    MessageBox.Show("Application installed! You may now run it from the Start menu.");
}

void iphm_DownloadProgressChanged(object sender, DownloadProgressChangedEventArgs e)
{
    // you can show percentage of task completed using e.ProgressPercentage
}

```

5. To attempt installation from your code, call the `InstallApplication` method. For example, if you named your class `MyInstaller`, you might call `InstallApplication` in the following way.

```
Dim installer As New MyInstaller()  
installer.InstallApplication(@"\myServer\myShare\myApp.application")  
MessageBox.Show("Installer object created.")
```

```
MyInstaller installer = new MyInstaller();  
installer.InstallApplication(@"\myServer\myShare\myApp.application");  
MessageBox.Show("Installer object created.");
```

## Next steps

A ClickOnce application can also add custom update logic, including a custom user interface to show during the update process. For more information, see [UpdateCheckInfo](#). A ClickOnce application can also suppress the standard Start menu entry, shortcut, and Add or Remove Programs entry by using a `<customUX>` element. For more information, see [<entryPoint> element](#) and [ShortcutAppld](#).

## See also

- [ClickOnce application manifest](#)
- [<entryPoint> element](#)

# Choose a ClickOnce update strategy

3/5/2021 • 6 minutes to read • [Edit Online](#)

ClickOnce can provide automatic application updates. A ClickOnce application periodically reads its deployment manifest file to see whether updates to the application are available. If available, the new version of the application is downloaded and run. For efficiency, only those files that have changed are downloaded.

When designing a ClickOnce application, you have to determine which strategy the application will use to check for available updates. There are three basic strategies that you can use: checking for updates on application startup, checking for updates after application startup (running in a background thread), or providing a user interface for updates.

In addition, you can determine how often the application will check for updates, and you can make updates required.

## NOTE

Application updates require network connectivity. If a network connection is not present, the application will run without checking for updates, regardless of the update strategy that you choose.

## NOTE

In .NET Framework 2.0 and .NET Framework 3.0, any time your application checks for updates, before or after startup, or by using the `<xref:System.Deployment.Application>` APIs, you must set `deploymentProvider` in the deployment manifest. The `deploymentProvider` element corresponds in Visual Studio to the **Update location** field on the **Updates** dialog box of the **Publish** tab. This rule is relaxed in .NET Framework 3.5. For more information, see [Deploying ClickOnce Applications For Testing and Production Servers without Resigning](#).

## Check for updates after application startup

By using this strategy, the application will attempt to locate and read the deployment manifest file in the background while the application is running. If an update is available, the next time that the user runs the application, he will be prompted to download and install the update.

This strategy works best for low-bandwidth network connections or for larger applications that might require lengthy downloads.

To enable this update strategy, click **After the application starts** in the **Choose when the application should check for updates** section of the **Application Updates** dialog box. Then specify an update interval in the section **Specify how frequently the application should check for updates**.

This is the same as changing the **Update** element in the deployment manifest as follows:

```
<!-- When to check for updates -->
<subscription>
  <update>
    <expiration maximumAge="6" unit="hours" />
  </update>
</subscription>
```



## Check for updates before application startup

The default strategy is to try to locate and read the deployment manifest file before the application starts. By using this strategy, the application will attempt to locate and read the deployment manifest file every time that the user starts the application. If an update is available, it will be downloaded and started; otherwise, the existing version of the application will be started.

This strategy works best for high-bandwidth network connections; the delay in starting the application may be unacceptably long over low-bandwidth connections.

To enable this update strategy, click **Before the application starts** in the **Choose when the application should check for updates** section of the **Application Updates** dialog box.

This is the same as changing the **Update** element in the deployment manifest as follows:

```
<!-- When to check for updates -->
<subscription>
  <update>
    <beforeApplicationStartup />
  </update>
</subscription>
```

### NOTE

For .NET 3.1 and newer applications, checking updates before the application starts is the only update option supported.

## Make updates required

There may be occasions when you want to require users to run an updated version of your application. For example, you might make a change to an external resource such as a Web service that would prevent the earlier version of your application from working correctly. In this case, you would want to mark your update as required and prevent users from running the earlier version.

### NOTE

Although you can require updates by using the other update strategies, checking **Before the application starts** is the only way to guarantee that an older version cannot be run. When the mandatory update is detected on startup, the user must either accept the update or close the application.

To mark an update as required, click **Specify a minimum required version for this application** in the **Application Updates** dialog box, and then specify the publish version (**Major, Minor, Build, Revision**), which specifies the lowest version number of the application that can be installed.

This is the same as setting the **minimumRequiredVersion** attribute of the **Deployment** element in the deployment manifest; for example:

```
<deployment install="true" minimumRequiredVersion="1.0.0.0">
```

## Specify update intervals

You can also specify how often the application checks for updates. To do this, specify that the application check for updates after startup as described in "Checking for Updates After Application Startup" earlier in this topic.

To specify the update interval, set the **Specify how frequently the application should check for updates**

properties in the **Application Updates** dialog box.

This is the same as setting the **maximumAge** and **unit** attributes of the **Update** element in the deployment manifest.

For example, you may want to check each time the application runs, or one time a week, or one time a month. If a network connection is not present at the specified time, the update check is performed the next time that the application runs.

## Provide a user interface for updates

When using this strategy, the application developer provides a user interface that enables the user to choose when or how often the application will check for updates. For example, you might provide a "Check for Updates Now" command, or an "Update Settings" dialog box that has choices for different update intervals. The ClickOnce deployment APIs provide a framework for programming your own update user interface. For more information, see the [System.Deployment.Application](#) namespace.

If your application uses deployment APIs to control its own update logic, you should block update checking as described in "Blocking Update Checking" in the following section.

This strategy works best when you need different update strategies for different users.

## Block update checking

It is also possible to prevent your application from ever checking for updates. For example, you might have a simple application that will never be updated, but you want to take advantage of the ease of installation provided by ClickOnce deployment.

You should also block update checking if your application uses deployment APIs to perform its own updates; see "Provide a user interface for updates" earlier in this topic.

To block update checking, clear the **The application should check for updates** check box in the Application Updates Dialog Box.

You can also block update checking by removing the `<Subscription>` tag from the deployment manifest.

## Permission elevation and updates

If a new version of a ClickOnce application requires a higher level of trust to run than the previous version, ClickOnce will prompt the user, asking him if he wants the application to be granted this higher level of trust. If the user declines to grant the higher trust level, the update will not install. ClickOnce will prompt the user to install the application again when it is next restarted. If the user declines to grant the higher level of trust at this point, and the update is not marked as required, the old version of the application will run. However, if the update is required, the application will not run again until the user accepts the higher trust level.

No prompting for trust levels will occur if you use Trusted Application Deployment. For more information, see [Trusted application deployment overview](#).

## See also

- [System.Deployment.Application](#)
- [ClickOnce security and deployment](#)
- [Choose a ClickOnce deployment strategy](#)
- [Secure ClickOnce applications](#)
- [How ClickOnce performs application updates](#)
- [How to: Manage updates for a ClickOnce application](#)

# How ClickOnce performs application updates

3/5/2021 • 2 minutes to read • [Edit Online](#)

ClickOnce uses the file version information specified in an application's deployment manifest to decide whether to update the application's files. After an update begins, ClickOnce uses a technique called *file patching* to avoid redundant downloading of application files.

## File patching

When updating an application, ClickOnce does not download all of the files for the new version of the application unless the files have changed. Instead, it compares the hash signatures of the files specified in the application manifest for the current application against the signatures in the manifest for the new version. If a file's signatures are different, ClickOnce downloads the new version. If the signatures match, the file has not changed from one version to the next. In this case, ClickOnce copies the existing file and uses it in the new version of the application. This approach prevents ClickOnce from having to download the entire application again, even if only one or two files have changed.

File patching also works for assemblies that are downloaded on demand using the [DownloadFileGroup](#) and [DownloadFileGroupAsync](#) methods.

If you use Visual Studio to compile your application, it will generate new hash signatures for all files whenever you rebuild the entire project. In this case, all assemblies will be downloaded to the client, although only a few assemblies may have changed.

File patching does not work for files that are marked as data and stored in the data directory. These are always downloaded regardless of the file's hash signature. For more information on the data directory, see [Access local and remote data in ClickOnce applications](#).

## See also

- [Choose a ClickOnce update strategy](#)
- [Choose a ClickOnce deployment strategy](#)

# How to: Check for application updates programmatically using the ClickOnce deployment API

4/2/2021 • 4 minutes to read • [Edit Online](#)

ClickOnce provides two ways to update an application once it is deployed. In the first method, you can configure the ClickOnce deployment to check automatically for updates at certain intervals. In the second method, you can write code that uses the [ApplicationDeployment](#) class to check for updates based on an event, such as a user request.

The following procedures show some code for performing a programmatic update and also describe how to configure your ClickOnce deployment to enable programmatic update checks.

In order to update a ClickOnce application programmatically, you must specify a location for updates. This is sometimes referred to as a deployment provider. For more information on setting this property, see [Choose a ClickOnce update strategy](#).

## NOTE

You can also use the technique described below to deploy your application from one location but update it from another. For more information, see [How to: Specify an alternate location for deployment updates](#).

## To check for updates programmatically

1. Create a new Windows Forms application using your preferred command-line or visual tools.
2. Create whatever button, menu item, or other user interface item you want your users to select to check for updates. From that item's event handler, call the following method to check for and install updates.

```
private void InstallUpdateSyncWithInfo()
{
    UpdateCheckInfo info = null;

    if (ApplicationDeployment.IsNetworkDeployed)
    {
        ApplicationDeployment ad = ApplicationDeployment.CurrentDeployment;

        try
        {
            info = ad.CheckForDetailedUpdate();
        }
        catch (DeploymentDownloadException dde)
        {
            MessageBox.Show("The new version of the application cannot be downloaded at this time. \n\nPlease check your network connection, or try again later. Error: " + dde.Message);
            return;
        }
        catch (InvalidDeploymentException ide)
        {
            MessageBox.Show("Cannot check for a new version of the application. The ClickOnce deployment is corrupt. Please redeploy the application and try again. Error: " + ide.Message);
            return;
        }
        catch (InvalidOperationException ioe)
        {
            return;
        }
    }
}
```

```

    {
        MessageBox.Show("This application cannot be updated. It is likely not a ClickOnce
application. Error: " + ioe.Message);
        return;
    }

    if (info.UpdateAvailable)
    {
        Boolean doUpdate = true;

        if (!info.IsUpdateRequired)
        {
            DialogResult dr = MessageBox.Show("An update is available. Would you like to update
the application now?", "Update Available", MessageBoxButtons.OKCancel);
            if (!(DialogResult.OK == dr))
            {
                doUpdate = false;
            }
        }
        else
        {
            // Display a message that the app MUST reboot. Display the minimum required version.
            MessageBox.Show("This application has detected a mandatory update from your current "
+
                "version to version " + info.MinimumRequiredVersion.ToString() +
                ". The application will now install the update and restart.",
                "Update Available", MessageBoxButtons.OK,
                MessageBoxIcon.Information);
        }

        if (doUpdate)
        {
            try
            {
                ad.Update();
                MessageBox.Show("The application has been upgraded, and will now restart.");
                Application.Restart();
            }
            catch (DeploymentDownloadException dde)
            {
                MessageBox.Show("Cannot install the latest version of the application. \n\nPlease
check your network connection, or try again later. Error: " + dde);
                return;
            }
        }
    }
}

```

```

public:
    void InstallUpdateSync()
    {
        if (ApplicationDeployment::IsNetworkDeployed)
        {
            bool isUpdateAvailable = false;
            ApplicationDeployment^ appDeployment =
                ApplicationDeployment::CurrentDeployment;

            try
            {
                isUpdateAvailable = appDeployment->CheckForUpdate();
            }
            catch (InvalidOperationException^ ex)
            {
                MessageBox::Show("The update check failed. Error: {0}",
                    ex->Message);
                return;
            }

            if (isUpdateAvailable)
            {
                try
                {
                    appDeployment->Update();
                    MessageBox::Show(
                        "The application has been upgraded, and will now " +
                        "restart.");
                    Application::Restart();
                }
                catch (Exception^ ex)
                {
                    MessageBox::Show("The update failed. Error: {0}",
                        ex->Message);
                    return;
                }
            }
        }
    }
}

```

```

Private Sub InstallUpdateSyncWithInfo()
    Dim info As UpdateCheckInfo = Nothing

    If (ApplicationDeployment.IsNetworkDeployed) Then
        Dim AD As ApplicationDeployment = ApplicationDeployment.CurrentDeployment

        Try
            info = AD.CheckForDetailedUpdate()
        Catch dde As DeploymentDownloadException
            MessageBox.Show("The new version of the application cannot be downloaded at this time. "
+ ControlChars.Lf & ControlChars.Lf & "Please check your network connection, or try again later.
Error: " + dde.Message)
            Return
        Catch ioe As InvalidOperationException
            MessageBox.Show("This application cannot be updated. It is likely not a ClickOnce
application. Error: " & ioe.Message)
            Return
        End Try

        If (info.UpdateAvailable) Then
            Dim doUpdate As Boolean = True

            If (Not info.IsUpdateRequired) Then
                Dim dr As DialogResult = MessageBox.Show("An update is available. Would you like to
update the application now?", "Update Available", MessageBoxButtons.OKCancel)
                If (Not System.Windows.Forms.DialogResult.OK = dr) Then
                    doUpdate = False
                End If
            Else
                ' Display a message that the app MUST reboot. Display the minimum required version.
                MessageBox.Show("This application has detected a mandatory update from your current "
& _
                    "version to version " & info.MinimumRequiredVersion.ToString() & _
                    ". The application will now install the update and restart.", _
                    "Update Available", MessageBoxButtons.OK, _
                    MessageBoxIcon.Information)
            End If

            If (doUpdate) Then
                Try
                    AD.Update()
                    MessageBox.Show("The application has been upgraded, and will now restart.")
                    Application.Restart()
                Catch dde As DeploymentDownloadException
                    MessageBox.Show("Cannot install the latest version of the application. " &
ControlChars.Lf & ControlChars.Lf & "Please check your network connection, or try again later.")
                    Return
                End Try
            End If
        End If
    End If
End Sub

```

3. Compile your application.

### Use Mage.exe to deploy an application that checks for updates programmatically

- Follow the instructions for deploying your application using Mage.exe as explained in [Walkthrough: Manually deploy a ClickOnce application](#). When calling Mage.exe to generate the deployment manifest, make sure to use the command-line switch `providerUrl`, and to specify the URL where ClickOnce should check for updates. If your application will update from `http://www.adatum.com/MyApp`, for example, your call to generate the deployment manifest might look like this:

```
mage -New Deployment -ToFile WindowsFormsApp1.application -Name "My App 1.0" -Version 1.0.0.0 -
AppManifest 1.0.0.0\MyApp.manifest -providerUrl http://www.adatum.com/MyApp/MyApp.application
```

### Using MageUI.exe to deploy an application that checks for updates programmatically

- Follow the instructions for deploying your application using Mage.exe as explained in [Walkthrough: Manually deploy a ClickOnce application](#). On the **Deployment Options** tab, set the **Start Location** field to the application manifest ClickOnce should check for updates. On the **Update Options** tab, clear the **This application should check for updates** check box.

## .NET Framework Security

Your application must have full-trust permissions to use programmatic updating.

## See also

- [How to: Specify an alternate location for deployment updates](#)
- [Choose a ClickOnce update strategy](#)
- [Publish ClickOnce applications](#)



# How to: Specify an alternate location for deployment updates

3/5/2021 • 2 minutes to read • [Edit Online](#)

You can install your ClickOnce application initially from a CD or a file share, but the application must check for periodic updates on the Web. You can specify an alternate location for updates in your deployment manifest so that your application can update itself from the Web after its initial installation.

## NOTE

Your application must be configured to install locally to use this feature. For more information, see [Walkthrough: Manually deploy a ClickOnce application](#). In addition, if you install a ClickOnce application from the network, setting an alternate location causes ClickOnce to use that location for both the initial installation and all subsequent updates. If you install your application locally (for example, from a CD), the initial installation is performed using the original media, and all subsequent updates will use the alternate location.

## Specify an alternate location for updates by using MageUI.exe (Windows Forms-based utility)

1. Open a .NET Framework command prompt and type:

```
mageui.exe
```

2. On the **File** menu, choose **Open** to open your application's deployment manifest.
3. Select the **Deployment Options** tab.
4. In the text box named **Launch Location**, enter the URL to the directory that will contain the deployment manifest for application updates.
5. Save the deployment manifest.

## Specify an alternate location for updates by using Mage.exe

1. Open a .NET Framework command prompt.
2. Set the update location using the following command. In this example, *HelloWorld.exe.application* is the path to your ClickOnce application manifest, which always has the .application extension, and `http://adatum.com/Update/Path` is the URL that ClickOnce will check for application updates.

```
Mage -Update HelloWorld.exe.application -ProviderUrl http://adatum.com/Update/Path
```

3. Save the file.

## NOTE

You now need to re-sign the file with *Mage.exe*. For more information, see [Walkthrough: Manually deploy a ClickOnce application](#).

## .NET Framework Security

If you install your application from an offline medium such as a CD, and the computer is online, ClickOnce first checks the URL specified by the `<deploymentProvider>` tag in the deployment manifest to determine if the update location contains a more recent version of the application. If it does, ClickOnce installs the application

directly from there, instead of from the initial installation directory, and the common language runtime (CLR) determines your application's trust level using `<deploymentProvider>`. If the computer is offline, or `<deploymentProvider>` is unreachable, ClickOnce installs from the CD, and the CLR grants trust based on the installation point; for a CD install, this means your application receives full trust. All subsequent updates will inherit that trust level.

All ClickOnce applications that use `<deploymentProvider>` should explicitly declare the permissions they need in their application manifest, so that the application does not receive different levels of trust on different computers.

## See also

- [Walkthrough: Manually deploy a ClickOnce application](#)
- [ClickOnce deployment manifest](#)
- [Secure ClickOnce applications](#)
- [Choose a ClickOnce update strategy](#)

# ClickOnce deployment samples and walkthroughs

3/5/2021 • 2 minutes to read • [Edit Online](#)

This section contains sample applications, example code, and step-by-step walkthroughs that illustrate the syntax, structure, and techniques used to deploy Windows Forms, WPF, and console applications.

The sample code is intended for instructional purposes, and should not be used in deployed solutions without modifications. In particular, security must be taken into greater consideration.

## ClickOnce deployment

| TOPIC  | DESCRIPTION   |
|--|---|
| <a href="#">Deploy a ClickOnce application manually</a>  | Explains how to use .NET Framework utilities to deploy your ClickOnce application.  |
| <a href="#">Download assemblies on demand with the ClickOnce deployment API</a>                    | Demonstrates how to mark certain assemblies in your application as "optional," and how to download them using classes in the <a href="#">System.Deployment.Application</a> namespace. |
| <a href="#">Download assemblies on demand with the ClickOnce deployment API using the designer</a> | Explains how to download application assemblies only when they are first used by the application.   |

## See also

- [Visual Studio walkthroughs](#)

# Troubleshoot ClickOnce deployments

3/5/2021 • 2 minutes to read • [Edit Online](#)

This topic helps you diagnose and resolve the most common issues with ClickOnce deployments.

In most cases, a ClickOnce application will download to a user's computer and run without any problems. There are some cases, however, where Web server or application configuration issues can cause unforeseen problems.

## Deployment considerations

[How to: Set a custom log file location for ClickOnce deployment errors](#)

Describes how to redirect all ClickOnce activation failures on a machine to a single log file.

[How to: Specify verbose log files for ClickOnce deployments](#)

Describes how to increase the detail that ClickOnce writes to log files.

[Server and client configuration issues in ClickOnce deployments](#)

Describes various issues with the configuration of your Web server that could cause difficulty downloading ClickOnce applications.

[Security, versioning, and manifest issues in ClickOnce deployments](#)

Describes miscellaneous issues surrounding ClickOnce deployments.

[Troubleshoot specific errors in ClickOnce deployments](#)

Describes specific scenarios in which a ClickOnce deployment cannot succeed, and provides steps for resolving them.

[Debug ClickOnce applications that use System.Deployment.Application](#)

Describes a technique for debugging ClickOnce applications that use System.Deployment.Application.

## See also

- [ClickOnce deployment manifest](#)
- [ClickOnce application manifest](#)
- [Visual Studio troubleshooting](#)

# How to: Set a custom log file location for ClickOnce deployment errors

3/5/2021 • 2 minutes to read • [Edit Online](#)

ClickOnce maintains activation log files for all deployments. These logs document any errors pertaining to installing and initializing a ClickOnce deployment. By default, ClickOnce creates one log file for each deployment activation. It stores these log files in the Temporary Internet Files folder. The log file for a deployment is displayed to the user when an activation failure occurs, and the user clicks **Details** in the resulting error dialog box.

You can change this behavior for a specific client by using Registry Editor (**regedit.exe**) to set a custom log file path. In this case, ClickOnce logs activation successes and failures for all deployments in a single file.

## Caution

If you use Registry Editor incorrectly, you may cause serious problems that may require you to reinstall your operating system. Use Registry Editor at your own risk.

## NOTE

You will need to truncate or delete the log file occasionally to prevent it from growing too large.

The following procedure describes how to set a custom log file location for a single client.

### To set a custom log file location

1. Open **Regedit.exe**.
2. Navigate to the node `HKCU\Software\Classes\Software\Microsoft\Windows\CurrentVersion\Deployment`.
3. Set the string value `LogFilePath` to the full path and filename of your preferred custom log location.

This location must be in a directory to which the user has write access. For example, on Windows Vista, create the following folder structure and set `LogFilePath` to `C:\Users\<username>\Documents\Logs\ClickOnce\installation.log`.

## See also

- [Troubleshoot ClickOnce deployments](#)

# How to: Specify verbose log files for ClickOnce deployments

3/5/2021 • 2 minutes to read • [Edit Online](#)

ClickOnce maintains activity log files for all deployments. These logs document details pertaining to installing, initializing, updating, and uninstalling a ClickOnce deployment. To increase the detail that ClickOnce writes to these log files, use Registry Editor (*regedit.exe*) to specify the verbosity level.

## Caution

If you use Registry Editor incorrectly, you may cause serious problems that may require you to reinstall the operating system. Use Registry Editor at your own risk.

The following procedure describes how to specify the verbosity level for ClickOnce log files for the current user. To reduce the level of verbosity, remove this registry value.

## To specify verbose log files

1. Open *Regedit.exe*.
2. Navigate to the node  
`HKEY_CURRENT_USER\Software\Classes\Software\Microsoft\Windows\CurrentVersion\Deployment`.
3. If necessary, create a new string value named `LogVerbosityLevel`.
4. Set the `LogVerbosityLevel` value to `1`.

## See also

- [Troubleshoot ClickOnce deployments](#)

# Server and client configuration issues in ClickOnce deployments

3/5/2021 • 8 minutes to read • [Edit Online](#)

If you use Internet Information Services (IIS) on Windows Server, and your deployment contains a file type that Windows does not recognize, such as a Microsoft Word file, IIS will refuse to transmit that file, and your deployment will not succeed.

Additionally, some Web servers and Web application software, such as ASP.NET, contain a list of files and file types that you cannot download. For example, ASP.NET prevents the download of all *Web.config* files. These files may contain sensitive information such as user names and passwords.

Although this restriction should cause no problems for downloading core ClickOnce files such as manifests and assemblies, this restriction may prevent you from downloading data files included as part of your ClickOnce application. In ASP.NET, you can resolve this error by removing the handler that prohibits downloading of such files from the IIS configuration manager. See the IIS server documentation for additional details.

Some Web servers might block files with extensions such as *.dll*, *.config*, and *.mdf*. Windows-based applications typically include files with some of these extensions. If a user attempts to run a ClickOnce application that accesses a blocked file on a Web server, an error will result. Rather than unblocking all file extensions, ClickOnce publishes every application file with a *.deploy* file extension by default. Therefore, the administrator only needs to configure the Web server to unblock the following three file extensions:

- *.application*
- *.manifest*
- *.deploy*

However, you can disable this option by clearing the **Use ".deploy" file extension** option on the [Publish Options Dialog Box](#), in which case you must configure the Web server to unblock all file extensions used in the application.

You will have to configure *.manifest*, *.application*, and *.deploy*, for example, if you are using IIS where you have not installed the .NET Framework, or if you are using another Web server (for example, Apache).

## ClickOnce and Secure Sockets Layer (SSL)

A ClickOnce application will work fine over SSL, except when Internet Explorer raises a prompt about the SSL certificate. The prompt can be raised when there is something wrong with the certificate, such as when the site names do not match or the certificate has expired. To make ClickOnce work over an SSL connection, make sure that the certificate is up-to-date, and that the certificate data matches the site data.

## ClickOnce and proxy authentication

ClickOnce provides support for Windows Integrated proxy authentication starting in .NET Framework 3.5. No specific machine.config directives are required. ClickOnce does not provide support for other authentication protocols such as Basic or Digest.

You can also apply a hotfix to .NET Framework 2.0 to enable this feature. For more information, see [FIX: Error message when you try to install a ClickOnce application that you created in the .NET Framework 2.0 onto a client computer that is configured to use a proxy server: "Proxy authentication required"](#).

For more information, see [<defaultProxy> element \(network settings\)](#).

## ClickOnce and Web browser compatibility

Currently, ClickOnce installations will launch only if the URL to the deployment manifest is opened using Internet Explorer. A deployment whose URL is launched from another application, such as Microsoft Office Outlook, will launch successfully only if Internet Explorer is set as the default Web browser.

### NOTE

Mozilla Firefox is supported if the deployment provider is not blank or the Microsoft .NET Framework Assistant extension is installed. This extension is packaged with .NET Framework 3.5 SP1. For XBAP support, the NPWPF plug-in is activated when needed.

## Activate ClickOnce applications through browser scripting

If you have developed a custom Web page that launches a ClickOnce application using Active Scripting, you may find that the application will not launch on some machines. Internet Explorer contains a setting called **Automatic prompting for file downloads**, which affects this behavior. This setting is available on the **Security** Tab in its **Options** menu that affects this behavior. It is called **Automatic prompting for file downloads**, and it is listed underneath the **Downloads** category. The property is set to **Enable** by default for intranet Web pages, and to **Disable** by default for Internet Web pages. When this setting is set to **Disable**, any attempt to activate a ClickOnce application programmatically (for example, by assigning its URL to the `document.location` property) will be blocked. Under this circumstance, users can launch applications only through a user-initiated download, for example, by clicking a hyperlink set to the application's URL.

## Additional server configuration issues

### Administrator permissions required

You must have Administrator permissions on the target server if you are publishing with HTTP. IIS requires this permissions level. If you are not publishing using HTTP, you only need write permission on the target path.

### Server authentication issues

When you publish to a remote server that has "Anonymous Access" turned off, you will receive the following warning:

"The files could not be downloaded from http://<remoteserver>/<myapplication>/. The remote server returned an error: (401) Unauthorized."

### NOTE

You can make NTLM (NT challenge-response) authentication work if the site prompts for credentials other than your default credentials, and, in the security dialog box, you click **OK** when you are prompted if you want to save the supplied credentials for future sessions. However, this workaround will not work for basic authentication.

## Use third-party Web servers

If you are deploying a ClickOnce application from a Web server other than IIS, you may experience a problem if the server is returning the incorrect content type for key ClickOnce files, such as the deployment manifest and application manifest. To resolve this problem, see your Web server's Help documentation about how to add new content types to the server, and make sure that all the file name extension mappings listed in the following table are in place.



| FILE NAME EXTENSION       | CONTENT TYPE                              |
|---------------------------|---|
| <code>.application</code> | <code>application/x-ms-application</code> |
| <code>.manifest</code>    | <code>application/x-ms-manifest</code>    |
| <code>.deploy</code>      | <code>application/octet-stream</code>     |
| <code>.msu</code>         | <code>application/octet-stream</code>     |
| <code>.msp</code>         | <code>application/octet-stream</code>     |

## ClickOnce and mapped drives

If you use Visual Studio to publish a ClickOnce application, you cannot specify a mapped drive as the installation location. However, you can modify the ClickOnce application to install from a mapped drive by using the Manifest Generator and Editor (Mage.exe and MageUI.exe). For more information, see [Mage.exe \(Manifest Generation and Editing Tool\)](#) and [MageUI.exe \(Manifest Generation and Editing Tool, Graphical Client\)](#).

## FTP protocol not supported for installing applications

ClickOnce supports installing applications from any HTTP 1.1 Web server or file server. FTP, the File Transfer Protocol, is not supported for installing applications. You can use FTP to publish applications only. The following table summarizes these differences:

| URL TYPE              | DESCRIPTION   |
|-----------------------|---|
| <code>ftp://</code>   | You can publish a ClickOnce application by using this protocol. |
| <code>http://</code>  | You can install a ClickOnce application by using this protocol. |
| <code>https://</code> | You can install a ClickOnce application by using this protocol. |
| <code>file://</code>  | You can install a ClickOnce application by using this protocol. |

## Windows XP SP2: Windows Firewall

By default, Windows XP SP2 enables the Windows Firewall. If you are developing your application on a computer that has Windows XP installed, you are still able to publish and run ClickOnce applications from the local server that is running IIS. However, you cannot access that server that is running IIS from another computer unless you open the Windows Firewall. See Windows Help for instructions on managing the Windows Firewall.

## Windows Server: Enable FrontPage server extensions

FrontPage Server Extensions from Microsoft is required for publishing applications to a Windows Web server that uses HTTP.

By default, Windows Server does not have FrontPage Server Extensions installed. If you want to use Visual Studio to publish to a Windows Server Web server that uses HTTP with FrontPage Server Extensions, you must install FrontPage Server Extensions first. You can perform the installation by using the Manage Your Server

administration tool in Windows Server.

## Windows Server: Locked-down content types

IIS on Windows Server 2003 locks down all file types except for certain known content types (for example, *.htm*, *.html*, *.txt*, and so on). To enable deployment of ClickOnce applications using this server, you need to change the IIS settings to allow downloading files of type *.application*, *.manifest*, and any other custom file types used by your application.

If you deploy using an IIS server, run *inetmgr.exe* and add new File Types for the default Web page:

- For the *.application* and *.manifest* extensions, the MIME type should be "application/x-ms-application."  
For other file types, the MIME type should be "application/octet-stream."
- If you create a MIME type with extension "<em>" and the MIME type "application/octet-stream," it will allow files of unblocked file type to be downloaded. (However, blocked file types such as *\*.aspx* and *\*.asmx* cannot be downloaded.)

For specific instructions on configuring MIME types on Windows Server, see [How to add a MIME type to a Web site or application](#).

## Content type mappings

When publishing over HTTP, the content type (also known as MIME type) for the *.application* file should be "application/x-ms-application." If you have .NET Framework 2.0 installed on the server, this will be set for you automatically. If this is not installed, then you need to create a MIME type association for the ClickOnce application vroot (or entire server).

If you deploy using an IIS server, run *inetmgr.exe* and add a new content type of "application/x-ms-application" for the *.application* extension.

## HTTP compression issues

With ClickOnce, you can perform downloads that use HTTP compression, a Web server technology that uses the GZIP algorithm to compress a data stream before sending the stream to the client. The client—in this case, ClickOnce—decompresses the stream before reading the files.

If you are using IIS, you can easily enable HTTP compression. However, when you enable HTTP compression, it is only enabled for certain file types—namely, HTML and text files. To enable compression for assemblies (*.dll*), XML (*.xml*), deployment manifests (*.application*), and application manifests (*.manifest*), you must add these file types to the list of types for IIS to compress. Until you add the file types to your deployment, only text and HTML files will be compressed.

For detailed instructions for IIS, see [How to specify additional document types for HTTP compression](#).

## See also

- [Troubleshoot ClickOnce deployments](#)
- [Choose a ClickOnce deployment strategy](#)
- [Application deployment prerequisites](#)

# Security, versioning, and manifest issues in ClickOnce deployments

3/5/2021 • 5 minutes to read • [Edit Online](#)

There are a variety of issues with ClickOnce security, application versioning, and manifest syntax and semantics that can cause a ClickOnce deployment not to succeed.

## ClickOnce and Windows Vista User Account Control

In Windows Vista, applications by default run as a standard user, even if the current user is logged in with an account that has administrator permissions. If an application must perform an action that requires administrator permissions, it tells the operating system, which then prompts the user to enter their administrator credentials. This feature, which is named User Account Control (UAC), prevents applications from making changes that may affect the entire operating system without a user's explicit approval. Windows applications declare that they require this permission elevation by specifying the `requestedExecutionLevel` attribute in the `trustInfo` section of their application manifest.

Due to the risk of exposing applications to security elevation attacks, ClickOnce applications cannot request permission elevation if UAC is enabled for the client. Any ClickOnce application that attempts to set its `requestedExecutionLevel` attribute to `requireAdministrator` or `highestAvailable` will not install on Windows Vista.

In some cases, your ClickOnce application may attempt to run with administrator permissions because of installer detection logic on Windows Vista. In this case, you can set the `requestedExecutionLevel` attribute in the application manifest to `asInvoker`. This will cause the application itself to run without elevation. Visual Studio 2008 automatically adds this attribute to all application manifests.

If you are developing an application that requires administrator permissions for the entire lifetime of the application, you should consider deploying the application by using Windows Installer (MSI) technology instead. For more information, see [Windows Installer basics](#).

## Online application quotas and partial trust applications

If your ClickOnce application runs online instead of through an installation, it must fit within the quota set aside for online applications. Also, a network application that runs in partial trust, such as with a restricted set of security permissions, cannot be larger than half of the quota size.

For more information, and instructions about how to change the online application quota, see [ClickOnce cache overview](#).

## Versioning issues

You may experience problems if you assign strong names to your assembly and increment the assembly version number to reflect an application update. Any assembly compiled with a reference to a strong-named assembly must itself be recompiled, or the assembly will try to reference the older version. The assembly will try this because the assembly is using the old version value in its bind request.

For example, say that you have a strong-named assembly in its own project with version 1.0.0.0. After compiling the assembly, you add it as a reference to the project that contains your main application. If you update the assembly, increment the version to 1.0.0.1, and try to deploy it without also recompiling the application, the

application will not be able to load the assembly at run time.

This error can occur only if you are editing your ClickOnce manifests manually; you should not experience this error if you generate your deployment using Visual Studio.

## Specify individual .NET Framework assemblies in the manifest

Your application will fail to load if you have manually edited a ClickOnce deployment to reference an older version of a .NET Framework assembly. For example, if you added a reference to the System.Net assembly for a version of the .NET Framework prior to the version specified in the manifest, then an error would occur. In general, you should not attempt to specify references to individual .NET Framework assemblies, as the version of the .NET Framework against which your application runs is specified as a dependency in the application manifest.

## Manifest parsing issues

The manifest files that are used by ClickOnce are XML files, and they must be both well-formed and valid: they must obey the XML syntax rules and only use elements and attributes defined in the relevant XML schema.

Something that can cause problems in a manifest file is selecting a name for your application that contains a special character, such as a single or double quotation mark. The application's name is part of its ClickOnce identity. ClickOnce currently does not parse identities that contain special characters. If your application fails to activate, make sure that you are using only alphabetical and numeric characters for the name, and attempt to deploy it again.

If you have manually edited your deployment or application manifests, you may have unintentionally corrupted them. Corrupted manifest will prevent a correct ClickOnce installation. You can debug such errors at run time by clicking **Details** on the **ClickOnce Error** dialog box, and reading the error message in the log. The log will list one of the following messages:

- A description of the syntax error, and the line number and character position where the error occurred.
- The name of an element or attribute used in violation of the manifest's schema. If you have added XML manually to your manifests, you will have to compare your additions to the manifest schemas. For more information, see [ClickOnce deployment manifest](#) and [ClickOnce application manifest](#).
- An ID conflict. Dependency references in deployment and application manifests must be unique in both their `name` and `publicKeyToken` attributes. If both attributes match between any two elements within a manifest, manifest parsing will not succeed.

## Precautions when manually changing manifests or applications

When you update an application manifest, you must re-sign both the application manifest and the deployment manifest. The deployment manifest contains a reference to the application manifest that includes that file's hash and its digital signature.

### Precautions with deployment provider usage

The ClickOnce deployment manifest has a `deploymentProvider` property which points to the full path of the location from where the application should be installed and serviced:

```
<deploymentProvider codebase="http://myserver/myapp.application" />
```

This path is set when ClickOnce creates the application and is compulsory for installed applications. The path points to the standard location where the ClickOnce installer will install the application from and search for updates. If you use the **xcopy** command to copy a ClickOnce application to a different location, but do not

change the `deploymentProvider` property, ClickOnce will still refer back to the original location when it tries to download the application.

If you want to move or copy an application, you must also update the `deploymentProvider` path, so that the client actually installs from the new location. Updating this path is mostly a concern if you have installed applications. For online applications that are always launched through the original URL, setting the `deploymentProvider` is optional. If `deploymentProvider` is set, it will be honored; otherwise, the URL used to start the application will be used as the base URL to download application files.

**NOTE**

Every time that you update the manifest you must also sign it again.

## See also

[Troubleshoot ClickOnce deployments](#) [Secure ClickOnce applications](#) [Choose a ClickOnce deployment strategy](#)

# Troubleshoot specific errors in ClickOnce deployments

3/5/2021 • 10 minutes to read • [Edit Online](#)

This article lists the following common errors that can occur when you deploy a ClickOnce application, and provides steps to resolve each problem.

## General errors

**When you try to locate an application file, nothing occurs, or XML renders in Internet Explorer, or you receive a Run or Save As dialog box**

This error is likely caused by content types (also known as MIME types) not being registered correctly on the server or client.

First, make sure that the server is configured to associate the *.application* extension with content type "application/x-ms-application."

If the server is configured correctly, check that the .NET Framework 2.0 is installed on your computer. If the .NET Framework 2.0 is installed, and you are still seeing this problem, try uninstalling and reinstalling the .NET Framework 2.0 to re-register the content type on the client.

**Error message says, "Unable to retrieve application. Files missing in deployment" or "Application download has been interrupted, check for network errors and try again later"**

This message indicates that one or more files being referenced by the ClickOnce manifests cannot be downloaded. The easiest way to debug this error is to try to download the URL that ClickOnce says it cannot download. Here are some possible causes:

- If the log file says "(403) Forbidden" or "(404) Not found," verify that the Web server is configured so that it does not block download of this file. For more information, see [Server and Client Configuration Issues in ClickOnce Deployments](#).
- If the *.config* file is being blocked by the server, see the section "Download error when you try to install a ClickOnce application that has a .config file" later in this article.
- Determine whether this error occurred because the `deploymentProvider` URL in the deployment manifest is pointing to a different location than the URL used for activation.
- Ensure that all files are present on the server; the ClickOnce log should tell you which file was not found.
- See whether there are network connectivity issues; you can receive this message if your client computer went offline during the download.

**Download error when you try to install a ClickOnce application that has a .config file**

By default, a Visual Basic Windows-based application includes an App.config file. There will be a problem when a user tries to install from a Web server that uses Windows Server 2003, because that operating system blocks the installation of *.config* files for security reasons. To enable the *.config* file to be installed, click **Use ".deploy" file extension** in the **Publish Options** dialog box.

You also must set the content types (also known as MIME types) appropriately for *.application*, *.manifest*, and *.deploy* files. For more information, see your Web server documentation.

For more information, see "Windows Server 2003: Locked-Down Content Types" in [Server and client configuration issues in ClickOnce deployments](#).

**Error message: "Application is improperly formatted;" Log file contains "XML signature is invalid"**

Ensure that you updated the manifest file and signed it again. Republish your application by using Visual Studio or use Mage to sign the application again.

**You updated your application on the server, but the client does not download the update**

This problem might be solved by completing one of the following tasks:

- Examine the `deploymentProvider` URL in the deployment manifest. Ensure that you are updating the bits in the same location that `deploymentProvider` points to.
- Verify the update interval in the deployment manifest. If this interval is set to a periodic interval, such as one time every six hours, ClickOnce will not scan for an update until this interval has passed. You can change the manifest to scan for an update every time that the application starts. Changing the update interval is a convenient option during development time to verify updates are being installed, but it slows down application activation.
- Try starting the application again on the Start menu. ClickOnce may have detected the update in the background, but will prompt you to install the bits on the next activation.

**During update you receive an error that has the following log entry: "The reference in the deployment does not match the identity defined in the application manifest"**

This error may occur because you have manually edited the deployment and application manifests, and have caused the description of the identity of an assembly in one manifest to become out of sync with the other. The identity of an assembly consists of its name, version, culture, and public key token. Examine the identity descriptions in your manifests, and correct any differences.

**First time activation from local disk or CD-ROM succeeds, but subsequent activation from Start Menu does not succeed**

ClickOnce uses the Deployment Provider URL to receive updates for the application. Verify that the location that the URL is pointing to is correct.

**Error: "Cannot start the application"**

This error message usually indicates that there is a problem installing this application into the ClickOnce store. Either the application has an error or the store is corrupted. The log file might tell you where the error occurred.

You should do the following:

- Verify that the identity of the deployment manifest, identity of application manifest, and identity of the main application EXE are all unique.
- Verify that your file paths are not longer than 100 characters. If your application contains file paths that are too long, you may exceed the limitations on the maximum path you can store. Try shortening the paths and reinstall.

**PrivatePath settings in application config file are not honored**

To use PrivatePath (Fusion probing paths), the application must request full trust permission. Try changing the application manifest to request full trust, and then try again.

**During uninstall a message appears saying, "Failed to uninstall application"**

This message usually indicates that the application has already been removed or the store is corrupted. After you click OK, the **Add/Remove Program** entry will be removed.

**During installation, a message appears that says that the platform dependencies are not installed**

You are missing a prerequisite in the GAC (global assembly cache) that the application needs in order to run.

## Publishing with Visual Studio

**Publishing in Visual Studio fails**

Ensure that you have the right to publish to the server that you are targeting. For example, if you are logged in to a terminal server computer as an ordinary user, not as an administrator, you probably will not have the rights

required to publish to the local Web server.

If you are publishing with a URL, ensure that the destination computer has FrontPage Server Extensions enabled.

**Error message: Unable to create the Web site '<site>'. The components for communicating with FrontPage Server Extensions are not installed.**

Ensure that you have the Microsoft Visual Studio Web Authoring Component installed on the machine that you are publishing from. For Express users, this component is not installed by default. For more information, see <http://go.microsoft.com/fwlink/?LinkId=102310>.

**Error message: Could not find file 'Microsoft.Windows.Common-Controls, Version=6.0.0.0, Culture=\*, PublicKeyToken=6595b64144ccf1df, ProcessorArchitecture=\*, Type=win32'**

This error message appears when you attempt to publish a WPF application with visual styles enabled. To resolve this issue, see [How to: Publish a WPF Application with Visual Styles Enabled](#).

## Using Mage

**You tried to sign with a certificate in your certificate store and a received blank message box**

In the **Signing** dialog box, you must:

- Select **Sign with a stored certificate**, and
- Select a certificate from the list; the first certificate is not the default selection.

**Clicking the "Don't Sign" button causes an exception**

This issue is a known bug. All ClickOnce manifests are required to be signed. Just select one of the signing options, and then click **OK**.

## Additional errors

The following table shows some common error messages that a client-computer user may receive when the user installs a ClickOnce application. Each error message is listed next to a description of the most probable cause for the error.

| ERROR MESSAGE  | DESCRIPTION  |
|--|--|
| Application cannot be started. Contact the application publisher.<br><br>Cannot start the application. Contact the application vendor for assistance.  | These are generic error messages that occur when the application cannot be started, and no other specific reason can be found. Frequently this means that the application is somehow corrupted, or that the ClickOnce store is corrupted.  |
| Cannot continue. The application is improperly formatted. Contact the application publisher for assistance.<br><br>Application validation did not succeed. Unable to continue.<br><br>Unable to retrieve application files. Files corrupt in deployment. | One of the manifest files in the deployment is syntactically not valid, or contains a hash that cannot be reconciled with the corresponding file. This error may also indicate that the manifest embedded inside an assembly is corrupted. Re-create your deployment and recompile your application, or find and fix the errors manually in your manifests.  |
| Cannot retrieve application. Authentication error.<br><br>Application installation did not succeed. Cannot locate applications files on the server. Contact the application publisher or your administrator for assistance.                              | One or more files in the deployment cannot be downloaded because you do not have permission to access them. This can be caused by a 403 Forbidden error being returned by a Web server, which may occur if one of the files in your deployment ends with an extension that makes the Web server treat it as a protected file. Also, a directory that contains one or more of the application's files might require a username and password in order to access. |



| ERROR MESSAGE  | DESCRIPTION  |
|--|--|
| Cannot download the application. The application is missing required files. Contact the application vendor or your system administrator for assistance.                              | One or more of the files listed in the application manifest cannot be found on the server. Check that you have uploaded all the deployment's dependent files, and try again.   |
| Application download did not succeed. Check your network connection, or contact your system administrator or network service provider.   | ClickOnce cannot establish a network connection to the server. Examine the server's availability and the state of your network.  |
| URLDownloadToCacheFile failed with HRESULT '<number>'. An error occurred trying to download '<file>'.  | <p>If a user has set Internet Explorer Advanced Security option "Warn if changing between secure and not secure mode" on the deployment target computer, and if the setup URL of the ClickOnce application being installed is redirected from a non-secure to a secure site (or vice-versa), the installation will fail because the Internet Explorer warning interrupts it.</p> <p>To resolve this error, you can do one of the following tasks:</p> <ul style="list-style-type: none"> <li>- Clear the security option.</li> <li>- Make sure the setup URL is not redirected in such a way that changes security modes.</li> <li>- Remove the redirection completely and point to the actual setup URL.</li> </ul> |
| An error has occurred writing to the hard disk. There might be insufficient space available on the disk. Contact the application vendor or your system administrator for assistance. | This may indicate insufficient disk space for storing the application, but it may also indicate a more general I/O error when you are trying to save the application files to the drive.   |
| Cannot start the application. There is not enough available space on the disk.   | The hard disk is full. Clear off space and try to run the application again.   |
| Too many deployed activations are attempting to load at once.  | ClickOnce limits the number of different applications that can start at the same time. This is largely to help protect against malicious attempts to instigate denial-of-service attacks against the local ClickOnce service; users who try to start the same application repeatedly, in rapid succession, will only end up with a single instance of the application.   |
| Shortcuts cannot be activated over the network.  | Shortcuts to a ClickOnce application can only be started on the local hard disk. They cannot be started by opening a URL that points to a shortcut file on a remote server.  |
| The application is too large to run online in partial trust. Contact the application vendor or your system administrator for assistance.   | An application that runs in partial trust cannot be larger than half of the size of the online application quota, which by default is 250 MB.  |

## See also

- [ClickOnce security and deployment](#)
- [Troubleshoot ClickOnce deployments](#)
- [Visual Studio troubleshooting](#)

# Debug ClickOnce applications that use System.Deployment.Application

3/5/2021 • 2 minutes to read • [Edit Online](#)

In Visual Studio, ClickOnce deployment allows you to configure how an application is updated. However, if you need to use and customize advanced ClickOnce deployment features, you will need to access the deployment object model provided by [System.Deployment.Application](#). You can use the [System.Deployment.Application](#) APIs for advanced tasks such as:

- Creating an "Update Now" option in your application
- Conditional, on-demand downloads of various application components
- Updates integrated directly into the application
- Guaranteeing that the client application is always up-to-date

Because the [System.Deployment.Application](#) APIs work only when an application is deployed with ClickOnce technology, the only way to debug them is to deploy the application using ClickOnce, attach to it, then debug it. It can be difficult to attach the debugger early enough, because this code often runs when the application starts up and executes before you can attach the debugger. A solution is to place breaks (or stops, for Visual Basic projects) before your update check code or on-demand code.

The recommended debugging technique is as follows:

1. Before you start, make sure the symbol (.pdb) and source files are archived.
2. Deploy version 1 of the application.
3. Create a new blank solution. From the **File** menu, click **New**, then **Project**. In the **New Project** dialog box, open the **Other Project Types** node, then select the **Visual Studio Solutions** folder. In the **Templates** pane, select **Blank Solution**.
4. Add the archived source location to the properties for this new solution. In **Solution Explorer**, right-click the solution node, then click **Properties**. In the **Property Pages** dialog box, select **Debug Source Files**, then add the directory of the archived source code. Otherwise, the debugger will find the out-of-date source files, since the source file paths are recorded in the .pdb file. If the debugger uses out-of-date source files, you see a message telling you that the source does not match.
5. Make sure the debugger can find the .pdb files. If you have deployed them with your application, the debugger finds them automatically. It always looks next to the assembly in question first. Otherwise, you will need to add the archive path to the **Symbol file (.pdb) locations** (to access this option, from the **Tools** menu, click **Options**, then open the **Debugging** node, and click **Symbols**).
6. Debug what happens between the `CheckForUpdate` and `Download / Update` method calls.

For example, the update code might be as follows:

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles  
Button1.Click  
    If My.Application.Deployment.IsNetworkDeployed Then  
  
        If (My.Application.Deployment.CheckForUpdate()) Then  
  
            My.Application.Deployment.Update()  
            Application.Restart()  
  
        End If  
  
    End If  
End Sub
```

7. Deploy version 2.
8. Attempt to attach the debugger to the version 1 application as it downloads an update for version 2. Alternatively you can use the `System.Diagnostics.Debugger.Break` method or simply `Stop` in Visual Basic. Of course, you should not leave these method calls in production code.

For example, assume you are developing a Windows Forms application, and you have an event handler for this method with the update logic in it. To debug this, simply attach before the button is pressed, then set a breakpoint (make sure that you open the appropriate archived file and set the breakpoint there).

Use the [IsNetworkDeployed](#) property to invoke the [System.Deployment.Application](#) APIs only when the application is deployed; the APIs should not be invoked during debugging in Visual Studio.

## See also

- [System.Deployment.Application](#)

# Application deployment prerequisites

3/5/2021 • 4 minutes to read • [Edit Online](#)

To have your application to install and run successfully, first install all components upon which your application is dependent onto the target computer. For example, most applications created using Visual Studio have a dependency on the .NET Framework. In this case, the correct version of the common language runtime must be present on the destination computer before the application is installed.

You can select these prerequisites in the **Prerequisites Dialog Box** and install the .NET Framework and any other redistributable as a part of your installation. This practice is known as *bootstrapping*. Visual Studio generates a Windows executable program named *Setup.exe*, also known as a *bootstrapper*. The bootstrapper is responsible for installing these prerequisites before your application runs. For more information about selecting these prerequisites, see [Prerequisites dialog box](#).

Each prerequisite is a bootstrapper package. A bootstrapper package is a group of directories and files containing the manifest files that describe how the prerequisites are installed. If your application prerequisites are not listed in the **Prerequisite Dialog Box**, you can create custom bootstrapper packages and add them to Visual Studio. Then you can select the prerequisites in the **Prerequisites Dialog Box**. For more information, see [Create bootstrapper packages](#).

By default, bootstrapping is enabled for ClickOnce deployment. The bootstrapper generated for ClickOnce deployment is signed. You can disable bootstrapping for a component, but only if you are sure that the correct version of the component is already installed on all target computers.

## Bootstrapping and ClickOnce deployment

Before installing an application on a client computer, ClickOnce examines the client to ensure that it has the requirements specified in the application manifest. These include the following requirements:

- The minimum required version of the common language runtime, which is specified as an assembly dependency in the application manifest.
- The minimum required version of the Windows operating system required by the application, as specified in the application manifest using the `<osVersionInfo>` element. (See [<dependency> element](#).)
- The minimum version of all assemblies that must be preinstalled in the global assembly cache (GAC), as specified by assembly dependency declarations in the assembly manifest.

ClickOnce can detect missing prerequisites, and you can install prerequisites by using a bootstrapper. For more information, see [How to: Install prerequisites with a ClickOnce application](#).

### NOTE

To change the values in the manifests generated by tools such as Visual Studio and *MageUI.exe*, you need to edit the application manifest in a text editor, and then re-sign both the application and deployment manifests. For more information, see [How to: Re-sign application and deployment manifests](#).

If you use Visual Studio and ClickOnce to deploy your application, the bootstrapper packages that are selected by default depend on the version of the .NET Framework in the solution. However, if you change the target .NET Framework version, you must update the options in the **Prerequisites Dialog Box** manually.

| TARGET .NET FRAMEWORK           | SELECTED BOOTSTRAPPER PACKAGES                               |
|---------------------------------|--|
| .NET Framework 4 Client Profile | .NET Framework 4 Client Profile<br><br>Windows Installer 3.1 |
| .NET Framework 4                | .NET Framework 4<br><br>Windows Installer 3.1                |

With ClickOnce deployment, the *Publish.htm* page generated by the ClickOnce Publish Wizard points either to a link that installs only the application, or to a link that installs both the application and the bootstrapped components.

If you generate the bootstrapper by using the ClickOnce Publish Wizard or the Publish Page in Visual Studio, the *Setup.exe* is automatically signed. However, if you want to use your customer's certificate to sign the bootstrapper, you can sign the file later.

## Bootstrapping and MSBuild

If you do not use Visual Studio, but rather compile your applications on the command line, you can create the ClickOnce bootstrapping application by using a Microsoft Build Engine (MSBuild) task. For more information, see [GenerateBootstrapper task](#).

As an alternative to bootstrapping, you can pre-deploy components using an electronic software distribution system, such as Microsoft Systems Management Server (SMS).

## Bootstrapper (Setup.exe) command-line arguments

The *Setup.exe* generated by Visual Studio and the MSBuild tasks supports the following set of command-line arguments. Any other arguments are forwarded to the application installer.

If you change any bootstrapper options, you must change the unsigned bootstrapper and then later sign the bootstrapper file.

| COMMAND-LINE ARGUMENT   | DESCRIPTION   |
|---|---|
| -?, -h, -help   | Displays a Help dialog box.   |
| -url, -componentsurl  | Shows the stored URL and components url for this set up.  |
| -url= <input type="text" value="location"/>                                       | Sets the URL where <i>Setup.exe</i> will look for the ClickOnce application.  |
| -componentsurl= <input type="text" value="location"/>                             | Sets the URL where <i>Setup.exe</i> will look for the dependencies, such as the .NET Framework.   |
| -homesite= <input type="text" value="true"/>   <input type="text" value="false"/> | When <input type="text" value="true"/> , downloads the dependencies from the preferred location on the vendor's site. This setting overrides the -componentsurl setting. When <input type="text" value="false"/> , downloads the dependencies from the URL specified by -componentsurl. |

## Operating system support

The Visual Studio bootstrapper is not supported on Windows Server 2008 Server Core or Windows Server 2008 R2 Server Core, as they provide a low-maintenance server environment with limited functionality. For example, the Server Core installation option only supports the .NET Framework 3.5 Server Core profile, which cannot run the Visual Studio features that depend on the full .NET Framework.

## See also

- [Choose a ClickOnce deployment strategy](#)
- [ClickOnce security and deployment](#)

# Deploy prerequisites for 64-bit applications

3/5/2021 • 2 minutes to read • [Edit Online](#)

ClickOnce deployment supports the installation of applications on 64-bit platforms. The target platforms are **x86** for 32-bit platforms, **x64** for machines supporting the AMD64 and EM64T instruction sets, and **Itanium** for the 64-bit Itanium processor.

## Prerequisites

The following table lists the redistributables that you can use as prerequisites for your 64-bit application's installation.

If you select a prerequisite that does not have 64-bit components, you may see a warning stating that the selected packages are not available for the 64-bit platform.

| REDISTRIBUTABLE   | X64 SUPPORT | IA64 SUPPORT |
|---|-------------|--------------|
| Visual Studio Tools for Office runtime                  | Yes         | No           |
| Visual C++ 2010 Runtime Libraries (IA64)                | No          | Yes          |
| Visual C++ 2010 Runtime Libraries (x64)                 | Yes         | No           |
| Microsoft .NET Framework 4 (x86 and x64)                | Yes         |              |
| Microsoft .NET Framework 4 Client Profile (x86 and x64) | Yes         |              |

## See also

- [Deploy applications, services, and components](#)
- [How to: Install prerequisites with a ClickOnce application](#)
- [64-bit applications](#)

# Create bootstrapper packages

3/5/2021 • 4 minutes to read • [Edit Online](#)

The Setup program is a generic installer that can be configured to detect and install redistributable components such as Windows Installer (.msi) files and executable programs. The installer is also known as a bootstrapper. It is programmed through a set of XML manifests that specify the metadata to manage the installation of the component. Each redistributable component, or prerequisite, that appears in the **Prerequisites** dialog box for ClickOnce is a bootstrapper package. A bootstrapper package is a group of directories and files that contain manifest files that describe how the prerequisite should be installed.

The bootstrapper first detects whether any of the prerequisites are already installed. If prerequisites are not installed, first the bootstrapper shows the license agreements. Second, after the end user accepts the license agreements, the installation begins for the prerequisites. Otherwise, if all the prerequisites are detected, the bootstrapper just starts the application installer.

## Create custom bootstrapper packages

You can generate the bootstrapper manifests by using the XML Editor in Visual Studio. To see an example of creating a bootstrapper package, see [Walkthrough: Create a custom bootstrapper with a privacy prompt](#).

To create a bootstrapper package, you have to create a product manifest and, for each localized version of a component, a package manifest as well.

- The product manifest, *product.xml*, contains any language-neutral metadata for the package. This contains metadata common to all the localized versions of the redistributable component. To create this file, see [How to: Create a Product Manifest](#).
- The package manifest, *package.xml*, contains language-specific metadata; it typically contains localized error messages. A component must have at least one package manifest for each localized version of that component. To create this file, see [How to: Create a Package Manifest](#).

After these files are created, put the product manifest file into a folder named for the custom bootstrapper. The package manifest file goes into a folder named for the locale. For example, if the package manifest file is for English redistribution, put the file into a folder called en. Repeat this process for each locale, such as ja for Japanese and de for German. The final custom bootstrapper package could have the following folder structure.

```
CustomBootstrapperPackage
  product.xml
  CustomBootstrapper.msi
  de
    eula.rtf
    package.xml
  en
    eula.rtf
    package.xml
  ja
    eula.rtf
    package.xml
```

Next, copy the redistributable files into the bootstrapper folder location. For more information, see [How to: Create a localized bootstrapper package](#).



```
*\Program Files (x86)\Microsoft SDKs\ClickOnce Bootstrapper*
```

or, for older versions of Visual Studio

```
*\Program Files\Microsoft Visual Studio 14.0\SDK\Bootstrapper\Packages*
```

or

```
*\Program Files (x86)\Microsoft Visual Studio 14.0\SDK\Bootstrapper\Packages*
```

You can also find the bootstrapper folder location from the **Path** value in the following registry key:

```
*HKLM\Software\Microsoft\GenericBootstrapper*
```

On 64-bit systems, use the following registry key:

```
*HKLM\Software\Wow6432Node\Microsoft\GenericBootstrapper*
```

Each redistributable component appears in its own subfolder under the packages directory. The product manifest and redistributable files must be put into this subfolder. Localized versions of the component and package manifests must be put in subfolders named according to Culture Name.

After these files are copied into the bootstrapper folder, the bootstrapper package automatically appears in the Visual Studio **Prerequisites** dialog box. If your custom bootstrapper package does not appear, close and then reopen the **Prerequisites** dialog box. For more information, see [Prerequisites dialog box](#).

The following table shows the properties that are automatically populated by the bootstrapper.

| PROPERTY                   | DESCRIPTION   |
|----------------------------|---|
| ApplicationName            | The name of the application.  |
| ProcessorArchitecture      | The processor and bits-per-word of the platform targeted by an executable. Values include the following: <ul style="list-style-type: none"><li>- Intel</li><li>- IA64</li><li>- AMD64</li></ul> |
| <a href="#">Version9x</a>  | The version number for Microsoft Windows 95, Windows 98, or Windows ME operating systems. The syntax of the version is Major.Minor.ServicePack.   |
| <a href="#">VersionNT</a>  | The version number for the Windows NT, Windows 2000, Windows XP, Windows Vista, Windows Server 2008, or Windows 7 operating systems. The syntax of the version is Major.Minor.ServicePack.      |
| <a href="#">VersionMSI</a> | The version of the Windows Installer assembly (msi.dll) to run during the installation.   |
| <a href="#">AdminUser</a>  | This property is set if the user has administrator privileges. Values are true or false.  |

| PROPERTY    | DESCRIPTION   |
|-------------|---|
| InstallMode | <p>The installation mode indicates where the component needs to be installed from. Values include the following:</p> <ul style="list-style-type: none"> <li>- HomeSite - prerequisites are installed from the vendor's Web site.</li> <li>- SpecificSite - prerequisites are installed from the location that you select.</li> <li>- SameSite - prerequisites are installed from the same location as the application.</li> </ul> |

## Separate redistributables from application installations

You can prevent your redistributable files from being deployed in Setup projects. To do this, create a redistributable list in the RedistList folder in your .NET Framework directory:

```
%ProgramFiles%\Microsoft.NET\RedistList
```

The redistributable list is an XML file that you should name using the following format: *<Company Name>. <Component Name>.RedistList.xml*. So, for example, if the component is called DataWidgets made by Acme, use *Acme.DataWidgets.RedistList.xml*. An example of the redistributable list's contents might resemble this:

```
<?xml version="1.0" encoding="UTF-8"?>
<FileList Redist="Acme.DataWidgets" >
<File AssemblyName="Acme.DataGrid" Version="1.0.0.0" PublicKeyToken="b03f5f7f11d50a3a" Culture="neutral"
ProcessorArchitecture="MSIL" InGAC="true" />
</FileList>
```

## See also

- [How to: Install prerequisites with a ClickOnce application](#)
- [Prerequisites dialog box](#)
- [Product and package schema reference](#)
- [Use the Visual Studio 2005 bootstrapper to kick-start your installation](#)

# How to: Create a product manifest

3/5/2021 • 2 minutes to read • [Edit Online](#)

To deploy prerequisites for your application, you can create a bootstrapper package. A bootstrapper package contains a single product manifest file but a package manifest for each locale. The package manifest contains localization-specific aspects of your package. This includes strings, end-user license agreements, and the language packs.

For more information about package manifests, see [How to: Create a package manifest](#).

## Create the product manifest

### To create the product manifest

1. Create a directory for the bootstrapper package. This example uses C:\package.
2. In Visual Studio, create a new XML file called *product.xml*, and save it to the C:\package folder.
3. Add the following XML to describe the XML namespace and product code for the package. Replace the product code with a unique identifier for the package.

```
<Product
  xmlns="http://schemas.microsoft.com/developer/2004/01/bootstrapper"
  ProductCode="Custom.Bootstrapper.Package">
```

4. Add XML to specify that the package has a dependency. This example uses a dependency on Microsoft Windows Installer 3.1.

```
<RelatedProducts>
  <DependsOnProduct Code="Microsoft.Windows.Installer.3.1" />
</RelatedProducts>
```

5. Add XML to list all the files that are in the bootstrapper package. This example uses the package file name *CorePackage.msi*.

```
<PackageFiles>
  <PackageFile Name="CorePackage.msi"/>
</PackageFiles>
```

6. Copy or move the *CorePackage.msi* file to the C:\package folder.
7. Add XML to install the package by using bootstrapper commands. The bootstrapper automatically adds the `/qn` flag to the *.msi* file, which will install silently. If the file is an *.exe*, the bootstrapper runs the *.exe* file by using the shell. The following XML shows no arguments to *CorePackage.msi*, but you can put command line argument into the `Arguments` attribute.

```
<Commands>
  <Command PackageFile="CorePackage.msi" Arguments="">
```

8. Add the following XML to check if this bootstrapper package is installed. Replace the product code with the GUID for the redistributable component.

```
<InstallChecks>
  <MsiProductCheck
    Property="IsMsiInstalled"
    Product="{XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}" />
</InstallChecks>
```

9. Add XML to change the bootstrapper behavior depending on if the bootstrapper component is already installed. If the component is installed, the bootstrapper package does not run. The following XML checks if the current user is an administrator because this component requires administrative privileges.

```
<InstallConditions>
  <BypassIf
    Property="IsMsiInstalled"
    Compare="ValueGreaterThan" Value="0" />
  <FailIf Property="AdminUser"
    Compare="ValueNotEqualTo" Value="True"
    String="NotAnAdmin" />
</InstallConditions>
```

10. Add XML to set exit codes if the installation is successful and if a reboot is necessary. The following XML demonstrates the Fail and FailReboot exit codes, which indicate that the bootstrapper will not continue installing packages.

```
<ExitCodes>
  <ExitCode Value="0" Result="Success" />
  <ExitCode Value="1641" Result="SuccessReboot" />
  <ExitCode Value="3010" Result="SuccessReboot" />
  <DefaultExitCode Result="Fail" String="GeneralFailure" />
</ExitCodes>
```

11. Add the following XML to end the section for bootstrapper commands.

```
</Command>
</Commands>
```

12. Move the *C:\package* folder to the Visual Studio bootstrapper directory. For Visual Studio 2010, this is the *\Program Files\Microsoft SDKs\Windows\v7.0A\Bootstrapper\Packages* directory.

## Example

The product manifest contains installation instructions for custom prerequisites.

```

<?xml version="1.0" encoding="utf-8" ?>
<Product
  xmlns="http://schemas.microsoft.com/developer/2004/01/bootstrapper"
  ProductCode="Custom.Bootstrapper.Package">

  <RelatedProducts>
    <DependsOnProduct Code="Microsoft.Windows.Installer.3.1" />
  </RelatedProducts>

  <PackageFiles>
    <PackageFile Name="CorePackage.msi"/>
  </PackageFiles>

  <InstallChecks>
    <MsiProductCheck Product="IsMsiInstalled"
      Property="{XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}"/>
  </InstallChecks>

  <Commands>
    <Command PackageFile="CorePackage.msi" Arguments="">

      <InstallConditions>
        <BypassIf Property="IsMsiInstalled"
          Compare="ValueGreaterThan" Value="0"/>
        <FailIf Property="AdminUser"
          Compare="ValueNotEqualTo" Value="True"
          String="NotAnAdmin"/>
      </InstallConditions>

      <ExitCodes>
        <ExitCode Value="0" Result="Success"/>
        <ExitCode Value="1641" Result="SuccessReboot"/>
        <ExitCode Value="3010" Result="SuccessReboot"/>
        <DefaultExitCode Result="Fail" String="GeneralFailure"/>
      </ExitCodes>
    </Command>
  </Commands>
</Product>

```

## See also

- [Product and package schema reference](#)

# How to: Create a package manifest

3/5/2021 • 2 minutes to read • [Edit Online](#)

To deploy prerequisites for your application, you can use a bootstrapper package. A bootstrapper package contains a single product manifest file but a package manifest for each locale. Shared functionality across different localized versions should go into the product manifest.

For more information about product manifests, see [How to: Create a product manifest](#).

## Create the package manifest

### To create the package manifest

1. Create a directory for the bootstrapper package. This example uses *C:\package*.
2. Create a subdirectory with the name of the locale, such as *en* for English.
3. In Visual Studio, create an XML file that is named *package.xml*, and save it to the *C:\package\en* folder.
4. Add XML to list the name of the bootstrapper package, the culture for this localized package manifest, and the optional license agreement. The following XML uses the variables `DisplayName` and `Culture`, which are defined in a later element.

```
<Package
  xmlns="http://schemas.microsoft.com/developer/2004/01/bootstrapper"
  Name="DisplayName"
  Culture="Culture"
  LicenseAgreement="eula.txt">
```

5. Add XML to list all the files that are in the locale-specific directory. The following XML uses a file that is named *eula.txt* that is applicable for the *en* locale.

```
<PackageFiles>
  <PackageFile Name="eula.txt"/>
</PackageFiles>
```

6. Add XML to define localizable strings for the bootstrapper package. The following XML adds error strings for the *en* locale.

```
<Strings>
  <String Name="DisplayName">Custom Bootstrapper Package</String>
  <String Name="CultureName">en</String>
  <String Name="NotAnAdmin">You must be an administrator to install
this package.</String>
  <String Name="GeneralFailure">A general error has occurred while
installing this package.</String>
</Strings>
```

7. Copy the *C:\package* folder to the Visual Studio bootstrapper directory. For Visual Studio 2010, this is the *\Program Files\Microsoft SDKs\Windows\v7.0A\Bootstrapper\Packages* directory.

## Example

The package manifest contains locale-specific information, such as error messages, software license terms, and

language packs.

```
<?xml version="1.0" encoding="utf-8" ?>
<Package
  xmlns="http://schemas.microsoft.com/developer/2004/01/bootstrapper"
  Name="DisplayName"
  Culture="Culture"
  LicenseAgreement="eula.txt">

  <PackageFiles>
    <PackageFile Name="eula.txt"/>
  </PackageFiles>

  <Strings>
    <String Name="DisplayName">Custom Bootstrapper Package</String>
    <String Name="Culture">en</String>
    <String Name="NotAnAdmin">You must be an administrator to install this package.</String>
    <String Name="GeneralFailure">A general error has occurred while
installing this package.</String>
  </Strings>
</Package>
```

## See also

- [Product and package schema reference](#)

# How to: Create a localized bootstrapper package

3/5/2021 • 2 minutes to read • [Edit Online](#)

After you create a bootstrapper package, you can create localized versions of the bootstrapper package by creating two more files for each locale: a software license terms file (such as a *eula.rtf*) and a package manifest (*package.xml*).

By default, Visual Studio 2010 includes localized bootstrapper packages only for .NET Framework 4, .NET Framework 4 Client Profile, F# Runtime 2.0, and F# Runtime 4.0. You can create localized packages for other bootstrappers by completing three steps.

1. Create a folder that is named after the locale name in `\Program Files\Microsoft SDKs\Windows\v7.0A\Bootstrapper\Packages\<BootstrapperPackageName>`.
2. Create a file that contains the software license terms for the bootstrapper package and put it in the new folder.
3. Create a package manifest named *package.xml*, update the strings and culture, and put the file in the new folder. If you have already created a bootstrapper of Visual Studio in the target language, you can copy the Visual Studio *package.xml* file and modify it in this step.

## NOTE

If you are using a Setup project to deploy applications, you can localize your application by changing the **Localization** property.

## NOTE

Your computer might show different names or locations for some of the Visual Studio user interface elements in this article. You may be using a different edition of Visual Studio or different environment settings. For more information, see [Personalize the IDE](#).

## To create a localized bootstrapper package

1. Create a folder that is named after the locale name.

On 32-bit computers, create the folder in the `\Program Files\Microsoft SDKs\Windows\v7.0A\Bootstrapper\Packages\<BootstrapperPackageName>` folder.

On 64-bit computers, create the folder in the `\Program Files (86)\Microsoft SDKs\Windows\v7.0A\Bootstrapper\Packages\<BootstrapperPackageName>` folder.

The following table shows the folder names that you can use to match a locale.

| LOCALE                | FOLDER NAME |
|-----------------------|-------------|
| Chinese (Simplified)  | zh-Hans     |
| Chinese (Traditional) | zh-Hant     |
| Czech                 | cs          |



| LOCALE              | FOLDER NAME |
|---------------------|-------------|
| German              | de          |
| English             | en          |
| Spanish             | es          |
| French              | fr          |
| Italian             | it          |
| Korean              | ko          |
| Japanese            | ja          |
| Polish              | pl          |
| Portuguese (Brazil) | pt-BR       |
| Russian             | ru          |
| Turkish             | tr          |

2. Create a file that contains the software license terms for the bootstrapper package and put it in the new folder.
3. Create a package manifest named *package.xml* and put it in the new folder. For more information, see [How to: Create a package manifest](#).
4. Update the `<Strings>` section of the package manifest so that the strings are in the correct language for the locale.
5. Change the `<String Name="Culture">` value to match the folder name.
6. Save the *package.xml* file.

### To create a bootstrapper package for .NET Framework 3.5 Service Pack 1 localized in French

1. Create a folder that is named *fr*. The folder name must match the locale name.

On 32-bit computers, create the folder in the `\Program Files\Microsoft SDKs\Windows\v7.0A\Bootstrapper\Packages\DotNetFX35SP1\` folder.

On 64-bit computers, create the folder in the `\Program Files (x86)\Microsoft SDKs\Windows\v7.0A\Bootstrapper\Packages\DotNetFX35SP1\` folder.

2. Put a localized version of the software license terms into the *fr* folder.
3. Copy the `\Program Files (x86)\Microsoft SDKs\Windows\v7.0A\Bootstrapper\Packages\DotNetFX35SP1\en\package.xml` file to the *fr* folder, and open the file in the XML Designer.
4. Update the `<Strings>` section of the package manifest so that the error strings are in French.
5. Change the `<String Name="Culture">` value to *fr*.
6. Save the *package.xml* file.

## See also

- [Create bootstrapper packages](#)
- [Application deployment prerequisites](#)
- [How to: Create a package manifest](#)

# Walkthrough: Create a custom bootstrapper with a privacy prompt

4/2/2021 • 8 minutes to read • [Edit Online](#)

You can configure ClickOnce applications to automatically update when assemblies with newer file versions and assembly versions become available. To make sure that your customers consent to this behavior, you can display a privacy prompt to them. Then, they can choose whether to grant permission to the application to update automatically. If the application is not allowed to update automatically, it does not install.

## NOTE

Your computer might show different names or locations for some of the Visual Studio user interface elements in this article. You may be using a different edition of Visual Studio or different environment settings. For more information, see [Personalize the IDE](#).

## Prerequisites

You need the following components to complete this walkthrough:

- Visual Studio 2010.

## Create an Update Consent dialog box

To display a privacy prompt, create an application that asks the reader to consent to automatic updates for the application.

### To create a consent dialog box

1. On the **File** menu, point to **New**, and then click **Project**.
2. In the **New Project** dialog box, click **Windows**, and then click **WindowsFormsApplication**.
3. For the **Name**, type **ConsentDialog**, and then click **OK**.
4. In the designer, click the form.
5. In the **Properties** window, change the **Text** property to **Update Consent Dialog**.
6. In the **Toolbox**, expand **All Windows Forms**, and drag a **Label** control to the form.
7. In the designer, click the label control.
8. In the **Properties** window, change the **Text** property under **Appearance** to the following:  
  
The application that you are about to install checks for the latest updates on the Web. By clicking on "I Agree", you authorize the application to check for and install updates automatically from the Internet.
9. In the **Toolbox**, drag a **Checkbox** control to the middle of the form.
10. In the **Properties** window, change the **Text** property under **Layout** to **I Agree**.
11. In the **Toolbox**, drag a **Button** control to the lower left of the form.
12. In the **Properties** window, change the **Text** property under **Layout** to **Proceed**.

13. In the **Properties** window, change the **(Name)** property under **Design** to **ProceedButton**.
14. In the **Toolbox**, drag a **Button** control to the bottom right of the form.
15. In the **Properties** window, change the **Text** property under **Layout** to **Cancel**.
16. In the **Properties** window, change the **(Name)** property under **Design** to **CancelButton**.
17. In the designer, double-click the **I Agree** checkbox to generate the **CheckedChanged** event handler.
18. In the Form1 code file, add the following code for the **CheckedChanged** event handler.

```
private void checkBox1_CheckedChanged(object sender, EventArgs e)
{
    ProceedButton.Enabled = !ProceedButton.Enabled;
}
```

```
Private Sub CheckBox1_CheckedChanged(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles CheckBox1.CheckedChanged
    ProceedButton.Enabled = Not ProceedButton.Enabled
End Sub
```

19. Update the class constructor to disable the **Proceed** button by default.

```
public Form1()
{
    InitializeComponent();
    ProceedButton.Enabled = false;
}
```

```
Public Sub New()
    InitializeComponent()
    ProceedButton.Enabled = False
End Sub
```

20. In the Form1 code file, add the following code for a Boolean variable to track if the end user has consented to online updates.

```
public bool accepted = false;
```

```
Public accepted As Boolean = False
```

21. In the designer, double-click the **Proceed** button to generate the **Click** event handler.
22. In the Form1 code file, add the following code to the **Click** event handler for the **Proceed** button.

```
private void ProceedButton_Click(object sender, EventArgs e)
{
    if (ProceedButton.Enabled)
    {
        accepted = true;
        this.Close();
    }
}
```

```
Private Sub ProceedButton_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
ProceedButton.Click
    If ProceedButton.Enabled Then
        accepted = True
        Me.Close()
    End If
End Sub
```

23. In the designer, double-click the **Cancel** button to generate the Click event handler.

24. In the Form1 code file, add the following code for the Click event handler for the **Cancel** button.

```
private void CancelButton_Click(object sender, EventArgs e)
{
    this.Close();
}
```

```
Private Sub CancelButton_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
CancelButton.Click
    Me.Close()
End Sub
```

25. Update the application to return an error if the end user does not consent to online updates.

For Visual Basic developers only:

- a. In **Solution Explorer**, click **ConsentDialog**.
- b. On the **Project** menu, click **Add Module**, and then click **Add**.
- c. In the *Module1.vb* code file, add the following code.

```
Module Module1

    Function Main() As Integer
        Application.EnableVisualStyles()
        Application.SetCompatibleTextRenderingDefault(False)
        Dim f As New Form1()
        Application.Run(f)
        If (Not f.accepted) Then
            Return -1
        Else
            Return 0
        End If
    End Function

End Module
```

- d. On the **Project** menu, click **ConsentDialog Properties**, and then click the **Application** tab.
- e. Uncheck **Enable application framework**.
- f. In the **Startup object** drop-down menu, select **Module1**.

#### NOTE

Disabling the application framework disables features such as Windows XP visual styles, application events, splash screen, single instance application, and more. For more information, see [Application Page, Project Designer \(Visual Basic\)](#).

For Visual C# developers only:

Open the *Program.cs* code file, and add the following code.

```
static int Main()
{
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Form1 f = new Form1();
    Application.Run(f);
    if (!f.accepted)
        return -1;
    else
        return 0;
}
```

26. On the **Build** menu, click **BuildSolution**.

## Create the custom bootstrapper package

To show the privacy prompt to end users, you can create a custom bootstrapper package for the Update Consent Dialog application and include it as a prerequisite in all of your ClickOnce applications.

This procedure demonstrates how to create a custom bootstrapper package by creating the following documents:

- A *product.xml* manifest file to describe the contents of the bootstrapper.
- A *package.xml* manifest file to list the localization-specific aspects of your package, such as strings and the software license terms.
- A document for the software license terms.

#### Step 1: To create the bootstrapper directory

1. Create a directory named **UpdateConsentDialog** in the *%PROGRAMFILES%\Microsoft SDKs\Windows\v7.0A\Bootstrapper\Packages*.

#### NOTE

You may need administrative privileges to create this folder.

2. In the *UpdateConsentDialog* directory, create a subdirectory named *en*.

#### NOTE

Create a new directory for each locale. For example, you can add subdirectories for the *fr* and *de* locales. These directories would contain the French and German strings and language packs, if necessary.

#### Step 2: To create the product.xml manifest file

1. Create a text file called *product.xml*.

2. In the *product.xml* file, add the following XML code. Make sure that you do not overwrite the existing XML code.

```
<Product
  xmlns="http://schemas.microsoft.com/developer/2004/01/bootstrapper"
  ProductCode="Microsoft.Sample.EULA">
  <!-- Defines the list of files to be copied on build. -->
  <PackageFiles CopyAllPackageFiles="false">
    <PackageFile Name="ConsentDialog.exe"/>
  </PackageFiles>

  <!-- Defines how to run the Setup package.-->
  <Commands >
    <Command PackageFile = "ConsentDialog.exe" Arguments=''>
      <ExitCodes>
        <ExitCode Value="0" Result="Success" />
        <ExitCode Value="-1" Result="Fail" String="AU_Unaccepted" />
        <DefaultExitCode Result="Fail"
          FormatMessageFromSystem="true" String="GeneralFailure" />
      </ExitCodes>
    </Command>
  </Commands>

</Product>
```

3. Save the file to the UpdateConsentDialog bootstrapper directory.

### Step 3: To create the package.xml manifest file and the software license terms

1. Create a text file called *package.xml*.
2. In the *package.xml* file, add the following XML code to define the locale and include the software license terms. Make sure that you do not overwrite the existing XML code.

```
<Package
  xmlns="http://schemas.microsoft.com/developer/2004/01/bootstrapper"
  Name="DisplayName"
  Culture="Culture"
  LicenseAgreement="eula.rtf">
  <PackageFiles>
    <PackageFile Name="eula.rtf"/>
  </PackageFiles>

  <!-- Defines a localizable string table for error messages. -->
  <Strings>
    <String Name="DisplayName">Update Consent Dialog</String>
    <String Name="Culture">en</String>
    <String Name="AU_Unaccepted">The automatic update agreement is not accepted.</String>
    <String Name="GeneralFailure">A failure occurred attempting to launch the setup.</String>
  </Strings>
</Package>
```

3. Save the file to the en subdirectory in the UpdateConsentDialog bootstrapper directory.
4. Create a document called *eula.rtf* for the software license terms.

#### NOTE

The software license terms should include information about licensing, warranties, liabilities, and local laws. These files should be locale-specific, so make sure that the file is saved in a format that supports MBCS or UNICODE characters. Consult your legal department about the content of the software license terms.

5. Save the document to the en subdirectory in the *UpdateConsentDialog* bootstrapper directory.
6. If necessary, create a new *package.xml* manifest file and a new *eula.rtf* document for the software license terms for each locale. For example, if you created subdirectories for the fr and de locales, create separate *package.xml* manifest files and software license terms and save them to the fr and de subdirectories.

## Set the Update Consent Application as a prerequisite

In Visual Studio, you can set the Update Consent application as a prerequisite.

### To set the Update Consent Application as a prerequisite

1. In **Solution Explorer**, click the name of your application that you want to deploy.
2. On the **Project** menu, click *ProjectNameProperties*.
3. Click the **Publish** page, and then click **Prerequisites**.
4. Select **Update Consent Dialog**.

#### NOTE

You may have to close and reopen Visual Studio to see the Update Consent Dialog in the Prerequisites Dialog Box.

5. Click **OK**.

## Create and test the Setup program

After you set the Update Consent application as a prerequisite, you can generate the installer and bootstrapper for your application.

### To create and test the Setup program by not clicking I agree

1. In **Solution Explorer**, click the name of your application that you want to deploy.
2. On the **Project** menu, click *ProjectNameProperties*.
3. Click the **Publish** page, and then click **Publish Now**.
4. If the publish output does not open automatically, navigate to the publish output.
5. Run the *Setup.exe* program.

The Setup program shows the Update Consent Dialog software license agreement.

6. Read the software license agreement, and then click **Accept**.

The Update Consent Dialog application appears and shows the following text: The application that you are about to install checks for the latest updates on the Web. By clicking on I Agree, you authorize the application to check for updates automatically on the Internet.

7. Close the application or click **Cancel**.

The application shows an error: An error occurred while installing system components for *ApplicationName*. Setup cannot continue until all system components have been successfully installed.

8. Click **Details** to show the following error message: Component Update Consent Dialog has failed to install with the following error message: "The automatic update agreement is not accepted." The following components failed to install: - Update Consent Dialog
9. Click **Close**.



To create and test the Setup program by clicking I agree

1. In **Solution Explorer**, click the name of your application that you want to deploy.
2. On the **Project** menu, click *ProjectNameProperties*.
3. Click the **Publish** page, and then click **Publish Now**.
4. If the publish output does not open automatically, navigate to the publish output.
5. Run the *Setup.exe* program.

The Setup program shows the Update Consent Dialog software license agreement.

6. Read the software license agreement, and then click **Accept**.

The Update Consent Dialog application appears and shows the following text: The application that you are about to install checks for the latest updates on the Web. By clicking on I Agree, you authorize the application to check for updates automatically on the Internet.

7. Click **I Agree**, and then click **Proceed**.

The application starts to install.

8. If the Application Install dialog box appears, click **Install**.

## See also

- [Application deployment prerequisites](#)
- [Create bootstrapper packages](#)
- [How to: Create a product manifest](#)
- [How to: Create a package manifest](#)
- [Product and package schema reference](#)

# Product and package schema reference

3/5/2021 • 3 minutes to read • [Edit Online](#)

A *product file* is an XML manifest that describes all of the external dependencies required by a ClickOnce application. Examples of external dependencies include the .NET Framework and the Microsoft Data Access Components (MDAC). A package file is similar to a product file but is used to install the culture-dependent components of a dependency, such as localized assemblies, license agreements, and documentation.

The product and packages file consists of either a top-level `Product` or `Package` element, each of which contains the following elements.

| ELEMENT   | DESCRIPTION   | ATTRIBUTES   |
|---|---|--|
| <a href="#">&lt;Product&gt; Element</a>         | Required top-level element for product files.   | None   |
| <a href="#">&lt;Package&gt; Element</a>         | Required top-level element for package files.   | <code>Culture</code><br><code>Name</code><br><code>EULA</code> |
| <a href="#">&lt;RelatedProducts&gt; Element</a> | Optional element for product files. The other products that this product either installs or depends upon.   | None   |
| <a href="#">&lt;InstallChecks&gt; Element</a>   | Required element. Lists the dependency checks to perform on the local computer during installation.   | None   |
| <a href="#">&lt;Commands&gt; Element</a>        | Required element. Executes one or more installation checks as described by <code>InstallChecks</code> , and denotes which package to install should the check fail. | None   |
| <a href="#">&lt;PackageFiles&gt; Element</a>    | Required element. Lists the packages that might be installed by this installation process.  | None   |
| <a href="#">&lt;Strings&gt; Element</a>         | Required element. Stores localized versions of the product name and error strings.  | None   |

## Remarks

The package schema is consumed by *Setup.exe*, a stub program generated by the MS Build bootstrapping task that contains little hard-coded logic of its own. The schema drives every aspect of the installation process.

`InstallChecks` the tests that setup.exe should perform for the existence of a given package. `PackageFiles` lists all of the packages that the setup process might have to install, should a given test fail. Each Command entry under Commands executes one of the tests described by `InstallChecks`, and specifies which `PackageFile` to

run should the test fail. You can use the `Strings` element to localize product names and error messages, so that you can use one single installation binary to install your application for any number of languages.

## Example

The following code example demonstrates a complete product file for installing the .NET Framework.

```
<?xml version="1.0" encoding="utf-8" ?>

<Product
  xmlns="http://schemas.microsoft.com/developer/2004/01/bootstrapper"
  ProductCode="Microsoft.Net.Framework.2.0"
>

  <RelatedProducts>
    <IncludesProduct Code="Microsoft.Windows.Installer.2.0" />
  </RelatedProducts>

  <!-- Defines list of files to be copied on build -->
  <PackageFiles>
    <PackageFile Name="instmsia.exe" HomeSite="InstMsiAExe"
      PublicKey="3082010A0282010100AA99BD39A81827F42B3D0B4C3F7C772EA7CBB5D18C0DC23A74D793B5E0A04B3F595ECE454F9A792
      9F149CC1A47EE55C2083E1220F855F2EE5FD3E0CA96BC30DEFE58C82732D08554E8F09110BBF32BBE19E5039B0B861DF3B0398CB8FD0
      B1D3C7326AC572BCA29A215908215E277A34052038B9DC270BA1FE934F6F335924E5583F8DA30B620DE5706B55A4206DE59CBF2DFA6B
      D154771192523D2CB6F9B1979DF6A5BF176057929FCC356CA8F440885558ACBC80F464B55CB8C96774A87E8A94106C7FF0DE96857637
      2C36957B443CF323A30DC1BE9D543262A79FE95DB226724C92FD034E3E6FB514986B83CD0255FD6EC9E036187A96840C7F8E203E6CF0
      50203010001"/>
    <PackageFile Name="WindowsInstaller-KB884016-v2-x86.exe" HomeSite="Msi30Exe"
      PublicKey="3082010A0282010100B22D8709B55CDF5599EB5262E7D3F4E34571A932BF94F20EE90DADFE9DC7046A584E9CA4D1D8444
      1FB647E0F65EEC817DA4DDBD9D650B40C565B6C16884BBF03EE504883EC4F88939A51E394197FFAB397A5CE606D9FDD4C9338BDCD345
      971E686CEE98399A096B8EAE0445B1342B93A484E5472F70896E400C482017643AF61C2DBFAE5C5F00213DDF835B40F0D5236467443B
      1A2CA9CDD7E99F1351177FB1526018ECFE0B804782A15FD72C66076910CE74FB218181B6989B4F12F211B66EACA91C7460DB91758715
      856866523D10232AE64A06FDA5295FDFBDD8D34F5C10C35A347D7E91B6AFA0F45B4E8321D7019BDD1F9E5641FEB8737EA6FD40D838FF
      D0203010001"/>
    <PackageFile Name="dotnetfx.exe" HomeSite="DotNetFXExe"
      PublicKey="3082010A0282010100B22D8709B55CDF5599EB5262E7D3F4E34571A932BF94F20EE90DADFE9DC7046A584E9CA4D1D8444
      1FB647E0F65EEC817DA4DDBD9D650B40C565B6C16884BBF03EE504883EC4F88939A51E394197FFAB397A5CE606D9FDD4C9338BDCD345
      971E686CEE98399A096B8EAE0445B1342B93A484E5472F70896E400C482017643AF61C2DBFAE5C5F00213DDF835B40F0D5236467443B
      1A2CA9CDD7E99F1351177FB1526018ECFE0B804782A15FD72C66076910CE74FB218181B6989B4F12F211B66EACA91C7460DB91758715
      856866523D10232AE64A06FDA5295FDFBDD8D34F5C10C35A347D7E91B6AFA0F45B4E8321D7019BDD1F9E5641FEB8737EA6FD40D838FF
      D0203010001"/>
    <PackageFile Name="dotnetchk.exe"/>
  </PackageFiles>

  <InstallChecks>
    <ExternalCheck Property="DotNetInstalled" PackageFile="dotnetchk.exe" />
    <RegistryCheck Property="IEVersion" Key="HKLM\Software\Microsoft\Internet Explorer" Value="Version"
  />
</InstallChecks>

  <!-- Defines how to invoke the setup for the .NET Framework redistributable -->
  <!-- TODO: Needs EstimatedTempSpace, LogFile, and an update of EstimatedDiskSpace -->
  <Commands Reboot="Defer">
    <Command PackageFile="instmsia.exe"
      Arguments= ' /q /c:"msiinst /delayrebootq"
      EstimatedInstallSeconds="20" >
    <InstallConditions>
      <BypassIf Property="VersionNT" Compare="ValueExists"/>
      <BypassIf Property="VersionMsi" Compare="VersionGreaterThanOrEqualTo" Value="2.0"/>
    </InstallConditions>
    <ExitCodes>
      <ExitCode Value="0" Result="SuccessReboot"/>
      <ExitCode Value="1641" Result="SuccessReboot"/>
      <ExitCode Value="3010" Result="SuccessReboot"/>
      <DefaultExitCode Result="Fail" FormatMessageFromSystem="true" String="GeneralFailure" />
    </ExitCodes>
  </Commands>
```

```

</Command>
<Command PackageFile="WindowsInstaller-KB884016-v2-x86.exe"
    Arguments= '/quiet /norestart'
    EstimatedInstallSeconds="20" >
    <InstallConditions>
        <BypassIf Property="Version9x" Compare="ValueExists"/>
        <BypassIf Property="VersionNT" Compare="VersionLessThan" Value="5.0.3"/>
        <BypassIf Property="VersionMsi" Compare="VersionGreaterThanOrEqual" Value="3.0"/>
        <FailIf Property="AdminUser" Compare="ValueEqualTo" Value="false" String="AdminRequired"/>
    </InstallConditions>
    <ExitCodes>
        <ExitCode Value="0" Result="Success"/>
        <ExitCode Value="1641" Result="SuccessReboot"/>
        <ExitCode Value="3010" Result="SuccessReboot"/>
        <DefaultExitCode Result="Fail" FormatMessageFromSystem="true" String="GeneralFailure" />
    </ExitCodes>
</Command>
<Command PackageFile="dotnetfx.exe"
    Arguments=' /q:a /c:"install /q /l"'
    EstimatedInstalledBytes="21000000"
    EstimatedInstallSeconds="300">

    <!-- These checks determine whether the package is to be installed -->
    <InstallConditions>
        <!-- Either of these properties indicates the .NET Framework is already installed -->
        <BypassIf Property="DotNetInstalled" Compare="ValueNotEqualTo" Value="0"/>

        <!-- Block install if user does not have admin privileges -->
        <FailIf Property="AdminUser" Compare="ValueEqualTo" Value="false" String="AdminRequired"/>

        <!-- Block install on Windows 95 -->
        <FailIf Property="Version9X" Compare="VersionLessThan" Value="4.10"
String="InvalidPlatformWin9x"/>

        <!-- Block install on Windows 2000 SP 2 or less -->
        <FailIf Property="VersionNT" Compare="VersionLessThan" Value="5.0.3"
String="InvalidPlatformWinNT"/>

        <!-- Block install if Internet Explorer 5.01 or greater is not present -->
        <FailIf Property="IEVersion" Compare="ValueNotExists" String="InvalidPlatformIE" />
        <FailIf Property="IEVersion" Compare="VersionLessThan" Value="5.01"
String="InvalidPlatformIE" />

        <!-- Block install if the platform is not x86 -->
        <FailIf Property="ProcessorArchitecture" Compare="ValueNotEqualTo" Value="Intel"
String="InvalidPlatformArchitecture" />
    </InstallConditions>

    <ExitCodes>
        <ExitCode Value="0" Result="Success"/>
        <ExitCode Value="3010" Result="SuccessReboot"/>
        <ExitCode Value="4097" Result="Fail" String="AdminRequired"/>
        <ExitCode Value="4098" Result="Fail" String="WindowsInstallerComponentFailure"/>
        <ExitCode Value="4099" Result="Fail" String="WindowsInstallerImproperInstall"/>
        <ExitCode Value="4101" Result="Fail" String="AnotherInstanceRunning"/>
        <ExitCode Value="4102" Result="Fail" String="OpenDatabaseFailure"/>
        <ExitCode Value="4113" Result="Fail" String="BetaNDPFailure"/>
        <DefaultExitCode Result="Fail" FormatMessageFromSystem="true" String="GeneralFailure" />
    </ExitCodes>

</Command>
</Commands>
</Product>

```

## See also

- [ClickOnce deployment manifest](#)

- [ClickOnce application manifest](#)

# <Product> element (bootstrapper)

3/5/2021 • 2 minutes to read • [Edit Online](#)

The `Product` element is the top-level XML element inside of a product file.

## Syntax

```
<Product
ProductCode
>
  <RelatedProducts>
    <IncludesProduct
      Code
    >
  </RelatedProducts>

  <InstallChecks>
    <AssemblyCheck
      Property
      Name
      PublicKeyToken
      Version
      Language
      ProcessorArchitecture
    />
    <RegistryCheck
      Property
      Key
      Value
    />
    <ExternalCheck
      PackageFile
      Property
      Arguments
      Log
    />
    <FileCheck
      Property
      FileName
      SearchPath
      SpecialFolder
      SearchDepth
    />
    <MsiProductCheck
      Property
      Product
      Feature
    />
    <RegistryFileCheck
      Property
      Key
      Value
      File
      SearchDepth
    />
  </InstallChecks>

  <Commands
    Reboot
  >
  <Command
```

```

    PackageFile
    Arguments
    EstimatedInstallSeconds
    EstimatedDiskBytes
    EstimatedTempBytes
    Log
  >
  <InstallConditions>
    <BypassIf
      Property
      Compare
      Value
      Schedule
    />
    <FailIf
      Property
      Compare
      Value
      String
      Schedule
    />
  </InstallConditions>
  <ExitCodes>
    <ExitCode
      Value
      Result
      String
    />
  </ExitCodes>
</Command>
</Commands>

<PackageFiles
  CopyAllComponents
>
  <PackageFile
    Name
    Path
    HomeSite
    PublicKey
  />
</PackageFiles>

<Schedules>
  <Schedule
    Name
  >
    <BuildList />
    <BeforePackage />
    <AfterPackage />
  </Schedule>
</Schedules>
</Package>

```

## Elements and attributes

The `Product` element is required in a product file. It has the following attribute.

| ATTRIBUTE                | DESCRIPTION                          |
|--------------------------|--------------------------------------|
| <code>ProductCode</code> | A unique identifier for the product. |

## Example

The following code example shows a complete product file for installing the .NET Framework.

```
<?xml version="1.0" encoding="utf-8" ?>

<Product
  xmlns="http://schemas.microsoft.com/developer/2004/01/bootstrapper"
  ProductCode="Microsoft.Net.Framework.2.0"
>

  <RelatedProducts>
    <IncludesProduct Code="Microsoft.Windows.Installer.2.0" />
  </RelatedProducts>

  <!-- Defines list of files to be copied on build -->
  <PackageFiles>
    <PackageFile Name="instmsia.exe" HomeSite="InstMsiAExe"
      PublicKey="3082010A0282010100AA99BD39A81827F42B3D0B4C3F7C772EA7CBB5D18C0DC23A74D793B5E0A04B3F595ECE454F9A792
      9F149CC1A47EE55C2083E1220F855F2EE5FD3E0CA96BC30DEFE58C82732D08554E8F09110BBF32BBE19E5039B0B861DF3B0398CB8FD0
      B1D3C7326AC572BCA29A215908215E277A34052038B9DC270BA1FE934F6F335924E5583F8DA30B620DE5706B55A4206DE59CBF2DFA6B
      D154771192523D2CB6F9B1979DF6A5BF176057929FCC356CA8F440885558ACBC80F464B55CB8C96774A87E8A94106C7FF0DE96857637
      2C36957B443CF323A30DC1BE9D543262A79FE95DB226724C92FD034E3E6FB514986B83CD0255FD6EC9E036187A96840C7F8E203E6CF0
      50203010001"/>
    <PackageFile Name="WindowsInstaller-KB884016-v2-x86.exe" HomeSite="Msi30Exe"
      PublicKey="3082010A0282010100B22D8709B55CDF5599EB5262E7D3F4E34571A932BF94F20EE90DADFE9DC7046A584E9CA4D1D8444
      1FB647E0F65EEC817DA4DDBD9D650B40C565B6C16884BBF03EE504883EC4F88939A51E394197FFAB397A5CE606D9FDD4C9338BDCD345
      971E686CEE98399A096B8EAE0445B1342B93A484E5472F70896E400C482017643AF61C2DBFAE5C5F00213DDF835B40F0D5236467443B
      1A2CA9CDD7E99F1351177FB1526018ECFE0B804782A15FD72C66076910CE74FB218181B6989B4F12F211B66EACA91C7460DB91758715
      856866523D10232AE64A06FDA5295FDFBDD8D34F5C10C35A347D7E91B6AFA0F45B4E8321D7019BDD1F9E5641FEB8737EA6FD40D838FF
      D0203010001"/>
    <PackageFile Name="dotnetfx.exe" HomeSite="DotNetFXExe"
      PublicKey="3082010A0282010100B22D8709B55CDF5599EB5262E7D3F4E34571A932BF94F20EE90DADFE9DC7046A584E9CA4D1D8444
      1FB647E0F65EEC817DA4DDBD9D650B40C565B6C16884BBF03EE504883EC4F88939A51E394197FFAB397A5CE606D9FDD4C9338BDCD345
      971E686CEE98399A096B8EAE0445B1342B93A484E5472F70896E400C482017643AF61C2DBFAE5C5F00213DDF835B40F0D5236467443B
      1A2CA9CDD7E99F1351177FB1526018ECFE0B804782A15FD72C66076910CE74FB218181B6989B4F12F211B66EACA91C7460DB91758715
      856866523D10232AE64A06FDA5295FDFBDD8D34F5C10C35A347D7E91B6AFA0F45B4E8321D7019BDD1F9E5641FEB8737EA6FD40D838FF
      D0203010001"/>
    <PackageFile Name="dotnetchk.exe"/>
  </PackageFiles>

  <InstallChecks>
    <ExternalCheck Property="DotNetInstalled" PackageFile="dotnetchk.exe" />
    <RegistryCheck Property="IEVersion" Key="HKLM\Software\Microsoft\Internet Explorer" Value="Version"
  />
</InstallChecks>

  <!-- Defines how to invoke the setup for the .NET Framework redistributable -->
  <!-- TODO: Needs EstimatedTempSpace, LogFile, and an update of EstimatedDiskSpace -->
  <Commands Reboot="Defer">
    <Command PackageFile="instmsia.exe"
      Arguments= ' /q /c:"msiinst /delayrebootq"'
      EstimatedInstallSeconds="20" >
    <InstallConditions>
      <BypassIf Property="VersionNT" Compare="ValueExists"/>
      <BypassIf Property="VersionMsi" Compare="VersionGreaterThanOrEqualTo" Value="2.0"/>
    </InstallConditions>
    <ExitCodes>
      <ExitCode Value="0" Result="SuccessReboot"/>
      <ExitCode Value="1641" Result="SuccessReboot"/>
      <ExitCode Value="3010" Result="SuccessReboot"/>
      <DefaultExitCode Result="Fail" FormatMessageFromSystem="true" String="GeneralFailure" />
    </ExitCodes>
  </Command>
  <Command PackageFile="WindowsInstaller-KB884016-v2-x86.exe"
    Arguments= '/quiet /norestart'
    EstimatedInstallSeconds="20" >
  <InstallConditions>
    <BypassIf Property="Version9x" Compare="ValueExists"/>
    <BypassIf Property="VersionNT" Compare="VersionLessThan" Value="5.0.3"/>
  </InstallConditions>
</Commands>
</Product>
```



```

        <BypassIf Property="VersionMsi" Compare="VersionGreaterThanOrEqualTo" Value="3.0"/>
        <FailIf Property="AdminUser" Compare="ValueEqualTo" Value="false" String="AdminRequired"/>
    </InstallConditions>
    <ExitCodes>
        <ExitCode Value="0" Result="Success"/>
        <ExitCode Value="1641" Result="SuccessReboot"/>
        <ExitCode Value="3010" Result="SuccessReboot"/>
        <DefaultExitCode Result="Fail" FormatMessageFromSystem="true" String="GeneralFailure" />
    </ExitCodes>
</Command>
<Command PackageFile="dotnetfx.exe"
    Arguments=' /q:a /c:"install /q /l"'
    EstimatedInstalledBytes="21000000"
    EstimatedInstallSeconds="300">

    <!-- These checks determine whether the package is to be installed -->
    <InstallConditions>
        <!-- Either of these properties indicates the .NET Framework is already installed -->
        <BypassIf Property="DotNetInstalled" Compare="ValueNotEqualTo" Value="0"/>

        <!-- Block install if user does not have admin privileges -->
        <FailIf Property="AdminUser" Compare="ValueEqualTo" Value="false" String="AdminRequired"/>

        <!-- Block install on Windows 95 -->
        <FailIf Property="Version9X" Compare="VersionLessThan" Value="4.10"
String="InvalidPlatformWin9x"/>

        <!-- Block install on Windows 2000 SP 2 or less -->
        <FailIf Property="VersionNT" Compare="VersionLessThan" Value="5.0.3"
String="InvalidPlatformWinNT"/>

        <!-- Block install if Internet Explorer 5.01 or greater is not present -->
        <FailIf Property="IEVersion" Compare="ValueNotExists" String="InvalidPlatformIE" />
        <FailIf Property="IEVersion" Compare="VersionLessThan" Value="5.01"
String="InvalidPlatformIE" />

        <!-- Block install if the platform is not x86 -->
        <FailIf Property="ProcessorArchitecture" Compare="ValueNotEqualTo" Value="Intel"
String="InvalidPlatformArchitecture" />
    </InstallConditions>

    <ExitCodes>
        <ExitCode Value="0" Result="Success"/>
        <ExitCode Value="3010" Result="SuccessReboot"/>
        <ExitCode Value="4097" Result="Fail" String="AdminRequired"/>
        <ExitCode Value="4098" Result="Fail" String="WindowsInstallerComponentFailure"/>
        <ExitCode Value="4099" Result="Fail" String="WindowsInstallerImproperInstall"/>
        <ExitCode Value="4101" Result="Fail" String="AnotherInstanceRunning"/>
        <ExitCode Value="4102" Result="Fail" String="OpenDatabaseFailure"/>
        <ExitCode Value="4113" Result="Fail" String="BetaNDPFailure"/>
        <DefaultExitCode Result="Fail" FormatMessageFromSystem="true" String="GeneralFailure" />
    </ExitCodes>

</Command>
</Commands>
</Product>

```

## See also

- [Product and package schema reference](#)

# <Package> element (bootstrapper)

3/5/2021 • 2 minutes to read • [Edit Online](#)

The `Package` element is the top-level XML element inside of a package file.

## Syntax

```
<Package
  Culture
  Name
  LicenseAgreement
>
  <InstallChecks>
    <AssemblyCheck
      Property
      Name
      PublicKeyToken
      Version
      Language
      ProcessorArchitecture
    />
    <RegistryCheck
      Property
      Key
      Value
    />
    <ExternalCheck
      PackageFile
      Property
      Arguments
      Log
    />
    <FileCheck
      Property
      FileName
      SearchPath
      SpecialFolder
      SearchDepth
    />
    <MsiProductCheck
      Property
      Product
      Feature
    />
    <RegistryFileCheck
      Property
      Key
      Value
      File
      SearchDepth
    />
  </InstallChecks>

  <Commands
    Reboot
  >
    <Command
      PackageFile
      Arguments
      EstimatedInstallSeconds
      EstimatedDiskBytes
```

```

        EstimatedTempBytes
        Log
    >
    <InstallConditions>
        <BypassIf
            Property
            Compare
            Value
            Schedule
        />
        <FailIf
            Property
            Compare
            Value
            String
            Schedule
        />
    </InstallConditions>
    <ExitCodes>
        <ExitCode
            Value
            Result
            String
        />
    </ExitCodes>
</Command>
</Commands>

<PackageFiles
    CopyAllComponents
>
    <PackageFile
        Name
        Path
        HomeSite
        PublicKey
    />
</PackageFiles>

<Strings>
    <String
        Name
    >
</String>
</Strings>

<Schedules>
    <Schedule
        Name
    >
        <BuildList />
        <BeforePackage />
        <AfterPackage />
    </Schedule>
</Schedules>
</Package>

```

## Elements and attributes

The `Package` element is required. It has the following attributes.

| ATTRIBUTE | DESCRIPTION |
|-----------|-------------|
|-----------|-------------|

| ATTRIBUTE                     | DESCRIPTION   |
|-------------------------------|---|
| <code>Culture</code>          | Required. Defines the culture for this package, which determines the language to be used. This attribute is a key into the <code>Strings</code> element, which lists culture-specific strings for product names and error messages during the installation.   |
| <code>Name</code>             | Required. The name of the package displayed to the developer within a tool such as Visual Studio. This attribute is a key into the <code>Strings</code> element, which should contain a <code>String</code> element with the <code>Name</code> and <code>Culture</code> properties set to match the <code>Name</code> and <code>Culture</code> properties of <code>Package</code> . |
| <code>LicenseAgreement</code> | Optional. Specifies the name of the file in the distribution package which contains the End-User License Agreement (EULA). This file can be either plain text ( <i>.txt</i> ) or Rich Text Format ( <i>.rtf</i> ).  |

## Example

The following code example shows a complete package file for redistributing the .NET Framework 2.0.

```

<?xml version="1.0" encoding="utf-8" ?>

<Package
  xmlns="http://schemas.microsoft.com/developer/2004/01/bootstrapper"
  Name="DisplayName"
  Culture="Culture"
  LicenseAgreement="eula.rtf"
>

  <PackageFiles>
    <PackageFile Name="eula.rtf"/>
  </PackageFiles>

  <!-- Defines a localizable string table for error messages-->
  <Strings>
    <String Name="DisplayName">.NET Framework 2.0</String>
    <String Name="Culture">en</String>
    <String Name="AdminRequired">Administrator permissions are required to install the .NET Framework
2.0. Contact your administrator.</String>
    <String Name="InvalidPlatformWin9x">Installation of the .NET Framework 2.0 is not supported on
Windows 95. Contact your application vendor.</String>
    <String Name="InvalidPlatformWinNT">Installation of the .NET Framework 2.0 is not supported on
Windows NT 4.0. Contact your application vendor.</String>
    <String Name="InvalidPlatformIE">Installation of the .NET Framework 2.0 requires Internet Explorer
5.01 or greater. Contact your application vendor.</String>
    <String Name="InvalidPlatformArchitecture">This version of the .NET Framework 2.0 is not supported
on a 64-bit operating system. Contact your application vendor.</String>
    <String Name="WindowsInstallerImproperInstall">Due to an error with Windows Installer, the
installation of the .NET Framework 2.0 cannot proceed.</String>
    <String Name="AnotherInstanceRunning">Another instance of setup is already running. The running
instance must complete before this setup can proceed.</String>
    <String Name="BetaNDPFailure">A beta version of the .NET Framework was detected on the computer.
Uninstall any previous beta versions of .NET Framework before continuing.</String>
    <String Name="GeneralFailure">A failure occurred attempting to install the .NET Framework 2.0.
</String>
    <String Name="DotNetFXExe">http://go.microsoft.com/fwlink/?LinkId=37283</String>
    <String Name="InstMsiAExe">http://go.microsoft.com/fwlink/?LinkId=37285</String>
    <String Name="Msi30Exe">http://go.microsoft.com/fwlink/?LinkId=37287</String>
  </Strings>

</Package>

```

## See also

- [Product and package schema reference](#)

# <RelatedProducts> element (bootstrapper)

3/5/2021 • 2 minutes to read • [Edit Online](#)

The `RelatedProducts` element defines other products that either depend upon or are included in the current product.

## Syntax

```
<RelatedProducts>
  <DependsOnProduct
    Code
  />
  <EitherProducts>
    <DependsOnProduct
      Code
    />
  </EitherProducts>
  <IncludesProduct
    Code
  />
</RelatedProducts>
```

## Elements and attributes

The `RelatedProducts` element is a child of the `Product` element. It has no attributes.

## DependsOnProduct

The `DependsOnProduct` element signifies that the current product depends upon the named product, and that the named product should be installed before the current one. It is a child of the `RelatedProducts` element. A `RelatedProducts` element might have one or more `DependsOnProduct` elements.

`DependsOnProduct` has the following attribute.

| ATTRIBUTE         | DESCRIPTION  |
|-------------------|--|
| <code>Code</code> | The code name of the included product, as specified by the <code>ProductCode</code> attribute of the <code>Product</code> element. For more information, see <a href="#">&lt;Product&gt; Element</a> . |

## EitherProducts

The `EitherProducts` element defines zero or more `DependsOnProduct` elements, and has no attributes. At least one `DependsOnProduct` in this set must be installed before the current product. A `RelatedProducts` element can have zero or more `EitherProducts` elements.

## IncludesProduct

The `IncludesProduct` element signifies that a product is included with the current install, and does not require a separate installation. It is a child of the `RelatedProducts` element. A `RelatedProducts` element might have one or more `IncludesProduct` elements.

`IncludesProduct` has the following attribute.

| ATTRIBUTE         | DESCRIPTION  |
|-------------------|--|
| <code>Code</code> | The code name of the included product, as specified by the <code>ProductCode</code> attribute of the <code>Product</code> element. For more information, see <a href="#">&lt;Product&gt; Element</a> . |

## Example

The following code example specifies that the Microsoft Installer is installed with the .NET Framework, and therefore will not need a separate installation.

```
<RelatedProducts>
  <IncludesProduct Code="Microsoft.Windows.Installer.2.0" />
</RelatedProducts>
```

## See also

- [<Product> element](#)

# <InstallChecks> element (bootstrapper)

3/5/2021 • 7 minutes to read • [Edit Online](#)

The `InstallChecks` element supports starting a variety of tests against the local computer to make sure that all of the appropriate prerequisites for an application have been installed.

## Syntax

```
<InstallChecks>
  <AssemblyCheck
    Property
    Name
    PublicKeyToken
    Version
    Language
    ProcessorArchitecture
  />
  <RegistryCheck
    Property
    Key
    Value
  />
  <ExternalCheck
    PackageFile
    Property
    Arguments
  />
  <FileCheck
    Property
    FileName
    SearchPath
    SpecialFolder
    SearchDepth
  />
  <MsiProductCheck
    Property
    Product
    Feature
  />
  <RegistryFileCheck
    Property
    Key
    Value
    FileName
    SearchDepth
  />
</InstallChecks>
```

## AssemblyCheck

This element is an optional child element of `InstallChecks`. For each instance of `AssemblyCheck`, the bootstrapper will make sure that the assembly identified by the element exists in the global assembly cache (GAC). It contains no elements, and has the following attributes.



| ATTRIBUTE                          | DESCRIPTION   |
|------------------------------------|---|
| <code>Property</code>              | Required. The name of the property to store the result. This property can be referenced from a test underneath the <code>InstallConditions</code> element, which is a child of the <code>Command</code> element. For more information, see <a href="#">&lt;Commands&gt; Element</a> . |
| <code>Name</code>                  | Required. The fully qualified name of the assembly to check.  |
| <code>PublicKeyToken</code>        | Required. The abbreviated form of the public key associated with this strongly named assembly. All assemblies stored in the GAC must have a name, a version, and a public key.  |
| <code>Version</code>               | Required. The version of the assembly.<br><br>The version number has the format <i>&lt;major version&gt;.&lt;minor version&gt;.&lt;build version&gt;.&lt;revision version&gt;</i> .   |
| <code>Language</code>              | Optional. The language of a localized assembly. Default is <code>neutral</code> .   |
| <code>ProcessorArchitecture</code> | Optional. The computer processor targeted by this installation. Default is <code>msil</code> .  |

## ExternalCheck

This element is an optional child element of `InstallChecks`. For each instance of `ExternalCheck`, the bootstrapper will execute the named external program in a separate process, and store its exit code in the property indicated by `Property`. `ExternalCheck` is useful for implementing complex dependency checks, or when the only way to check for the existence of a component is to instantiate it.

`ExternalCheck` contains no elements, and has the following attributes.

| ATTRIBUTE                | DESCRIPTION   |
|--------------------------|---|
| <code>Property</code>    | Required. The name of the property to store the result. This property can be referenced from a test underneath the <code>InstallConditions</code> element, which is a child of the <code>Command</code> element. For more information, see <a href="#">&lt;Commands&gt; Element</a> . |
| <code>PackageFile</code> | Required. The external program to execute. The program must be part of the setup distribution package.  |
| <code>Arguments</code>   | Optional. Supplies command-line arguments to the executable named by <code>PackageFile</code> .   |

## FileCheck

This element is an optional child element of `InstallChecks`. For each instance of `FileCheck`, the bootstrapper will determine whether the named file exists, and return the version number of the file. If the file does not have a version number, the bootstrapper sets the property named by `Property` to 0. If the file does not exist, `Property` is not set to any value.

`FileCheck` contains no elements, and has the following attributes.

| ATTRIBUTE                  | DESCRIPTION  |
|----------------------------|--|
| <code>Property</code>      | Required. The name of the property to store the result. This property can be referenced from a test underneath the <code>InstallConditions</code> element, which is a child of the <code>Command</code> element. For more information, see <a href="#">&lt;Commands&gt; Element</a> .  |
| <code>FileName</code>      | Required. The name of the file to find.  |
| <code>SearchPath</code>    | Required. The disk or folder in which to look for the file. This must be a relative path if <code>SpecialFolder</code> is assigned; otherwise, it must be an absolute path.  |
| <code>SpecialFolder</code> | <p>Optional. A folder that has special significance either to Windows or to ClickOnce. The default is to interpret <code>SearchPath</code> as an absolute path. Valid values include the following:</p> <ul style="list-style-type: none"><li><code>AppDataFolder</code> . The application data folder for this ClickOnce application; specific to the current user.</li><li><code>CommonAppDataFolder</code> . The application data folder used by all users.</li><li><code>CommonFilesFolder</code> . The Common Files folder for the current user.</li><li><code>LocalDataAppFolder</code> . The data folder for non-roaming applications.</li><li><code>ProgramFilesFolder</code> . The standard Program Files folder for 32-bit applications.</li><li><code>StartupFolder</code> . The folder that contains all applications launched at system startup.</li><li><code>SystemFolder</code> . The folder that contains 32-bit system DLLs.</li><li><code>WindowsFolder</code> . The folder that contains the Windows system installation.</li><li><code>WindowsVolume</code> . The drive or partition that contains the Windows system installation.</li></ul> |
| <code>SearchDepth</code>   | Optional. The depth at which to search sub-folders for the named file. The search is depth-first. The default is 0, which restricts the search to the top-level folder specified by <code>SpecialFolder</code> and <code>SearchPath</code> .   |

## MsiProductCheck

This element is an optional child element of `InstallChecks` . For each instance of `MsiProductCheck` , the bootstrapper checks to see whether the specified Microsoft Windows Installer installation has run until it is completed. The property value is set depending on the state of that installed product. A positive value indicates the product is installed, 0 or -1 indicates it is not installed. (Please see the Windows Installer SDK function

MsiQueryFeatureState for more information.) . If Windows Installer is not installed on the computer, `Property` is not set.

`MsiProductCheck` contains no elements, and has the following attributes.

| ATTRIBUTE             | DESCRIPTION   |
|-----------------------|---|
| <code>Property</code> | Required. The name of the property to store the result. This property can be referenced from a test underneath the <code>InstallConditions</code> element, which is a child of the <code>Command</code> element. For more information, see <a href="#">&lt;Commands&gt; Element</a> . |
| <code>Product</code>  | Required. The GUID for the installed product.   |
| <code>Feature</code>  | Optional. The GUID for a specific feature of the installed application.   |

## RegistryCheck

This element is an optional child element of `InstallChecks` . For each instance of `RegistryCheck` , the bootstrapper checks to see whether the specified registry key exists, or whether it has the indicated value.

`RegistryCheck` contains no elements, and has the following attributes.

| ATTRIBUTE             | DESCRIPTION   |
|-----------------------|---|
| <code>Property</code> | Required. The name of the property to store the result. This property can be referenced from a test underneath the <code>InstallConditions</code> element, which is a child of the <code>Command</code> element. For more information, see <a href="#">&lt;Commands&gt; Element</a> . |
| <code>Key</code>      | Required. The name of the registry key.   |
| <code>Value</code>    | Optional. The name of the registry value to retrieve. The default is to return the text of the default value. <code>Value</code> must be either a String or a DWORD.  |

## RegistryFileCheck

This element is an optional child element of `InstallChecks` . For each instance of `RegistryFileCheck` , the bootstrapper retrieves the version of the specified file, first attempting to retrieve the path to the file from the specified registry key. This is particularly useful if you want to look up a file in a directory specified as a value in the registry.

`RegistryFileCheck` contains no elements, and has the following attributes.

| ATTRIBUTE             | DESCRIPTION   |
|-----------------------|---|
| <code>Property</code> | Required. The name of the property to store the result. This property can be referenced from a test underneath the <code>InstallConditions</code> element, which is a child of the <code>Command</code> element. For more information, see <a href="#">&lt;Commands&gt; Element</a> . |

| ATTRIBUTE                | DESCRIPTION   |
|--------------------------|---|
| <code>Key</code>         | Required. The name of the registry key. Its value is interpreted as the path to a file, unless the <code>File</code> attribute is set. If this key does not exist, <code>Property</code> is not set.  |
| <code>Value</code>       | Optional. The name of the registry value to retrieve. The default is to return the text of the default value. <code>Value</code> must be a String.  |
| <code>FileName</code>    | Optional. The name of a file. If specified, the value obtained from the registry key is assumed to be a directory path, and this name is appended to it. If not specified, the value returned from the registry is assumed to be the full path to a file. |
| <code>SearchDepth</code> | Optional. The depth at which to search sub-folders for the named file. The search is depth-first. The default is 0, which restricts the search to the top-level folder specified by the registry key's value.   |

## Remarks

While the elements underneath `InstallChecks` define the tests to run, they do not execute them. To execute the tests, you must create `Command` elements underneath the `Commands` element.

## Example

The following code example demonstrates the `InstallChecks` element as it is used in the product file for the .NET Framework.

```
<InstallChecks>
  <ExternalCheck Property="DotNetInstalled" PackageFile="dotnetchk.exe" />
  <RegistryCheck Property="IEVersion" Key="HKLM\Software\Microsoft\Internet Explorer" Value="Version" />
</InstallChecks>
```

## InstallConditions

When `InstallChecks` are evaluated, they produce properties. The properties are then used by `InstallConditions` to determine whether a package should install, bypass, or fail. The following table lists the `InstallConditions`:

| CONDITION             | DESCRIPTION   |
|-----------------------|---|
| <code>FailIf</code>   | If any <code>FailIf</code> condition evaluates to true, the package will fail. The rest of the conditions will not be evaluated.          |
| <code>BypassIf</code> | If any <code>BypassIf</code> condition evaluates to true, the package will be bypassed. The rest of the conditions will not be evaluated. |

## Predefined properties

The following table lists the `BypassIf` and `FailIf` elements:

| PROPERTY                 | NOTES   | POSSIBLE VALUES   |
|--------------------------|---|---|
| <code>Version9X</code>   | Version number of a Windows 9X operating system.  | 4.10 = Windows 98   |
| <code>VersionNT</code>   | Version number of a Windows NT-based operating system.  | Major.Minor.ServicePack<br>5.0 = Windows 2000<br>5.1.0 = Windows XP<br>5.1.2 = Windows XP Professional SP2<br>5.2.0 = Windows Server 2003 |
| <code>VersionNT64</code> | Version number of a 64-bit Windows NT-based operating system.                                 | Same as mentioned earlier.  |
| <code>VersionMsi</code>  | Version number of the Windows Installer service.  | 2.0 = Windows Installer 2.0   |
| <code>AdminUser</code>   | Specifies whether a user has administrator privileges on a Windows NT-based operating system. | 0 = no administrator privileges<br>1 = administrator privileges   |

For example, to block installation on a computer running Windows 95, use code such as the following:

```
<!-- Block install on Windows 95 -->
<FailIf Property="Version9X" Compare="VersionLessThan" Value="4.10" String="InvalidPlatform"/>
```

To skip running install checks if a `FailIf` or `BypassIf` condition is met, use the `BeforeInstallChecks` attribute. For example:

```
<!-- Block install and do not evaluate install checks if user does not have admin privileges -->
<FailIf Property="AdminUser" Compare="ValueEqualTo" Value="false" String="AdminRequired"
BeforeInstallChecks="true"/>
```

#### NOTE

The `BeforeInstallChecks` attribute is supported starting with the Visual Studio 2019 Update 9 release.

## See also

- [<Commands> element](#)
- [Product and package schema reference](#)

# <Commands> element (bootstrapper)

3/5/2021 • 5 minutes to read • [Edit Online](#)

The `Commands` element implements tests described by the elements underneath the `InstallChecks` element, and declares which package the ClickOnce bootstrapper should install if the test fails.

## Syntax

```
<Commands
  Reboot
>
  <Command
    PackageFile
    Arguments
    EstimatedInstallSeconds
    EstimatedDiskBytes
    EstimatedTempBytes
    Log
  >
    <InstallConditions>
      <BypassIf
        Property
        Compare
        Value
        Schedule
      />
      <FailIf
        Property
        Compare
        Value
        String
        Schedule
      />
    </InstallConditions>
    <ExitCodes>
      <ExitCode
        Value
        Result
        String
      />
    </ExitCodes>
  </Command>
</Commands>
```

## Elements and attributes

The `Commands` element is required. The element has the following attribute.

| ATTRIBUTE | DESCRIPTION |
|-----------|-------------|
|-----------|-------------|

| ATTRIBUTE | DESCRIPTION   |
|-----------|---|
| Reboot    | <p>Optional. Determines whether the system should restart if any of the packages return a restart exit code. The following list shows the valid values:</p> <p>Defer . The restart is deferred until some future time.</p> <p>Immediate . Causes an immediate restart if one of the packages returned a restart exit code.</p> <p>None . Causes any restart requests to be ignored.</p> <p>The default is Immediate .</p> |

## Command

The `Command` element is a child element of the `Commands` element. A `Commands` element can have one or more `Command` elements. The element has the following attributes.

| ATTRIBUTE               | DESCRIPTION   |
|-------------------------|---|
| PackageFile             | Required. The name of the package to install should one or more of the conditions specified by <code>InstallConditions</code> return false. The package must be defined in the same file by using a <code>PackageFile</code> element.   |
| Arguments               | Optional. A set of command line arguments to pass into the package file.  |
| EstimatedInstallSeconds | Optional. The estimated time, in seconds, it will take to install the package. This value determines the size of the progress bar the bootstrapper displays to the user. The default is 0, in which case no time estimate is specified.   |
| EstimatedDiskBytes      | Optional. The estimated amount of disk space, in bytes, that the package will occupy after the installation is finished. This value is used in hard disk space requirements that the bootstrapper displays to the user. The default is 0, in which case the bootstrapper does not display any hard disk space requirements. |
| EstimatedTempBytes      | Optional. The estimated amount of temporary disk space, in bytes, that the package will require.  |
| Log                     | Optional. The path to the log file that the package generates, relative to the root directory of the package.   |

## InstallConditions

The `InstallConditions` element is a child of the `Command` element. Each `Command` element can have at most one `InstallConditions` element. If no `InstallConditions` element exists, the package specified by `Condition` will always run.

## BypassIf

The `BypassIf` element is a child of the `InstallConditions` element, and describes a positive condition under which the command should not be executed. Each `InstallConditions` element can have zero or more `BypassIf` elements.

`BypassIf` has the following attributes.

| ATTRIBUTE             | DESCRIPTION   |
|-----------------------|---|
| <code>Property</code> | Required. The name of the property to test. The property must previously have been defined by a child of the <code>InstallChecks</code> element. For more information, see <a href="#">&lt;InstallChecks&gt; Element</a> .  |
| <code>Compare</code>  | Required. The type of comparison to perform. The following list shows the valid values:<br><br><code>ValueEqualTo</code> , <code>ValueNotEqualTo</code> , <code>ValueGreaterThan</code> , <code>ValueGreaterThanOrEqualTo</code> , <code>ValueLessThan</code> , <code>ValueLessThanOrEqualTo</code> , <code>VersionEqualTo</code> , <code>VersionNotEqualTo</code> , <code>VersionGreaterThan</code> , <code>VersionGreaterThanOrEqualTo</code> , <code>VersionLessThan</code> , <code>VersionLessThanOrEqualTo</code> , <code>ValueExists</code> , <code>ValueNotExists</code> |
| <code>Value</code>    | Required. The value to compare with the property.   |
| <code>Schedule</code> | Optional. The name of a <code>Schedule</code> tag that defines when this rule should be evaluated.  |

## FailIf

The `FailIf` element is a child of the `InstallConditions` element, and describes a positive condition under which the installation should stop. Each `InstallConditions` element can have zero or more `FailIf` elements.

`FailIf` has the following attributes.

| ATTRIBUTE             | DESCRIPTION   |
|-----------------------|---|
| <code>Property</code> | Required. The name of the property to test. The property must previously have been defined by a child of the <code>InstallChecks</code> element. For more information, see <a href="#">&lt;InstallChecks&gt; Element</a> .  |
| <code>Compare</code>  | Required. The type of comparison to perform. The following list shows the valid values:<br><br><code>ValueEqualTo</code> , <code>ValueNotEqualTo</code> , <code>ValueGreaterThan</code> , <code>ValueGreaterThanOrEqualTo</code> , <code>ValueLessThan</code> , <code>ValueLessThanOrEqualTo</code> , <code>VersionEqualTo</code> , <code>VersionNotEqualTo</code> , <code>VersionGreaterThan</code> , <code>VersionGreaterThanOrEqualTo</code> , <code>VersionLessThan</code> , <code>VersionLessThanOrEqualTo</code> , <code>ValueExists</code> , <code>ValueNotExists</code> |
| <code>Value</code>    | Required. The value to compare with the property.   |



| ATTRIBUTE             | DESCRIPTION  |
|-----------------------|--|
| <code>String</code>   | Optional. The text to display to the user upon failure.  |
| <code>Schedule</code> | Optional. The name of a <code>Schedule</code> tag that defines when this rule should be evaluated. |

## ExitCodes

The `ExitCodes` element is a child of the `Command` element. The `ExitCodes` element contains one or more `ExitCode` elements, which determine what the installation should do in response to an exit code from a package. There can be one optional `ExitCode` element underneath a `Command` element. `ExitCodes` has no attributes.

## ExitCode

The `ExitCode` element is a child of the `ExitCodes` element. The `ExitCode` element determines what the installation should do in response to an exit code from a package. `ExitCode` contains no child elements, and has the following attributes.

| ATTRIBUTE                            | DESCRIPTION   |
|--------------------------------------|---|
| <code>Value</code>                   | Required. The exit code value to which this <code>ExitCode</code> element applies.  |
| <code>Result</code>                  | <p>Required. How the installation should react to this exit code. The following list shows the valid values:</p> <ul style="list-style-type: none"> <li><code>Success</code> . Flags the package as successfully installed.</li> <li><code>SuccessReboot</code> . Flags the package as successfully installed, and instructs the system to restart.</li> <li><code>Fail</code> . Flags the package as failed.</li> <li><code>FailReboot</code> . Flags the package as failed, and instructs the system to restart.</li> </ul> |
| <code>String</code>                  | Optional. The value to display to the user in response to this exit code.   |
| <code>FormatMessageFromSystem</code> | Optional. Determines whether to use the system-provided error message corresponding to the exit code, or use the value provided in <code>String</code> . Valid values are <code>true</code> , which means to use the system-provided error, and <code>false</code> , which means to use the string provided by <code>String</code> . The default is <code>false</code> . If this property is <code>false</code> , but <code>String</code> is not set, the system-provided error will be used.                                 |

## Example

The following code example defines commands for installing the .NET Framework 2.0.

```
<Commands Reboot="Immediate">
  <Command PackageFile="instmsia.exe"
```

```

        Arguments= ' /q /c:"msiinst /delayrebootq" '
        EstimatedInstallSeconds="20" >
<InstallConditions>
    <BypassIf Property="VersionNT" Compare="ValueExists"/>
    BypassIf Property="VersionMsi" Compare="VersionGreaterThanOrEqualTo" Value="2.0"/>
</InstallConditions>
<ExitCodes>
    <ExitCode Value="0" Result="SuccessReboot"/>
    <ExitCode Value="1641" Result="SuccessReboot"/>
    <ExitCode Value="3010" Result="SuccessReboot"/>
    <DefaultExitCode Result="Fail" FormatMessageFromSystem="true" String="GeneralFailure" />
</ExitCodes>
</Command>
<Command PackageFile="WindowsInstaller-KB884016-v2-x86.exe"
    Arguments= '/quiet /norestart'
    EstimatedInstallSeconds="20" >
<InstallConditions>
    <BypassIf Property="Version9x" Compare="ValueExists"/>
    <BypassIf Property="VersionNT" Compare="VersionLessThan" Value="5.0.3"/>
    <BypassIf Property="VersionMsi" Compare="VersionGreaterThanOrEqualTo" Value="3.0"/>
    <FailIf Property="AdminUser" Compare="ValueEqualTo" Value="false" String="AdminRequired"/>
</InstallConditions>
<ExitCodes>
    <ExitCode Value="0" Result="Success"/>
    <ExitCode Value="1641" Result="SuccessReboot"/>
    <ExitCode Value="3010" Result="SuccessReboot"/>
    <DefaultExitCode Result="Fail" FormatMessageFromSystem="true" String="GeneralFailure" />
</ExitCodes>
</Command>
<Command PackageFile="dotnetfx.exe"
    Arguments=' /q:a /c:"install /q /l"'
    EstimatedInstalledBytes="21000000"
    EstimatedInstallSeconds="300">

<!-- These checks determine whether the package is to be installed -->
<InstallConditions>
    <!-- Either of these properties indicates the .NET Framework is already installed -->
    <BypassIf Property="DotNetInstalled" Compare="ValueNotEqualTo" Value="0"/>

    <!-- Block install if user does not have adminpermissions -->
    <FailIf Property="AdminUser" Compare="ValueEqualTo" Value="false" String="AdminRequired"/>

    <!-- Block install on Windows 95 -->
    <FailIf Property="Version9X" Compare="VersionLessThan" Value="4.10"
String="InvalidPlatformWin9x"/>

    <!-- Block install on Windows 2000 SP 2 or less -->
    <FailIf Property="VersionNT" Compare="VersionLessThan" Value="5.0.3"
String="InvalidPlatformWinNT"/>

    <!-- Block install if Internet Explorer 5.01 or later is not present -->
    <FailIf Property="IEVersion" Compare="ValueNotExists" String="InvalidPlatformIE" />
    <FailIf Property="IEVersion" Compare="VersionLessThan" Value="5.01" String="InvalidPlatformIE"
/>

    <!-- Block install if the operating system does not support x86 -->
    <FailIf Property="ProcessorArchitecture" Compare="ValueNotEqualTo" Value="Intel"
String="InvalidPlatformArchitecture" />
</InstallConditions>

<ExitCodes>
    <ExitCode Value="0" Result="Success"/>
    <ExitCode Value="3010" Result="SuccessReboot"/>
    <ExitCode Value="4097" Result="Fail" String="AdminRequired"/>
    <ExitCode Value="4098" Result="Fail" String="WindowsInstallerComponentFailure"/>
    <ExitCode Value="4099" Result="Fail" String="WindowsInstallerImproperInstall"/>
    <ExitCode Value="4101" Result="Fail" String="AnotherInstanceRunning"/>
    <ExitCode Value="4102" Result="Fail" String="OpenDatabaseFailure"/>
    <ExitCode Value="4113" Result="Fail" String="BetaNDPFailure"/>

```

```
        <DefaultExitCode Result="Fail" FormatMessageFromSystem="true" String="GeneralFailure" />
    </ExitCodes>

    </Command>
</Commands>
```

## See also

- [Product and package schema reference](#)
- [<InstallChecks> element](#)

# <PackageFiles> element (bootstrapper)

3/5/2021 • 2 minutes to read • [Edit Online](#)

The `PackageFiles` element contains `PackageFile` elements, which define the installation packages executed as a result of the `Command` element.

## Syntax

```
<PackageFiles
  CopyAllPackageFiles
>
  <PackageFile
    Name
    HomeSite
    CopyOnBuild
    PublicKey
    Hash
  />
</PackageFiles>
```

## Elements and attributes

The `PackageFiles` element has the following attribute.

| ATTRIBUTE                        | DESCRIPTION   |
|----------------------------------|---|
| <code>CopyAllPackageFiles</code> | <p>Optional. If set to <code>false</code>, the installer will only download files referenced from the <code>Command</code> element. If set to <code>true</code>, all files will be downloaded.</p> <p>If set to <code>IfNotHomeSite</code>, the installer will behave the same as if <code>False</code> if <code>ComponentsLocation</code> is set to <code>HomeSite</code>, and otherwise will behave the same as if <code>True</code>. This setting can be useful to allow packages that are themselves bootstrappers to execute their own behavior in a HomeSite scenario.</p> <p>The default is <code>true</code>.</p> |

## PackageFile

The `PackageFile` element is a child of the `PackageFiles` element. A `PackageFiles` element must have at least one `PackageFile` element.

`PackageFile` has the following attributes.

| ATTRIBUTE | DESCRIPTION |
|-----------|-------------|
|-----------|-------------|

| ATTRIBUTE                | DESCRIPTION   |
|--------------------------|---|
| <code>Name</code>        | Required. The name of the package file. This is the name that the <code>Command</code> element will reference when it defines the conditions under which a package installs. This value is also used as a key into the <code>Strings</code> table to retrieve the localized name that tools such as Visual Studio will use to describe the package. |
| <code>HomeSite</code>    | Optional. The location of the package on the remote server, if it is not included with the installer.   |
| <code>CopyOnBuild</code> | Optional. Specifies whether the bootstrapper should copy the package file onto the disk at build time. The default is true.   |
| <code>PublicKey</code>   | The encrypted public key of the package's certificate signer. Required if <code>HomeSite</code> is used; otherwise, optional.   |
| <code>Hash</code>        | Optional. An SHA1 hash of the package file. This is used to verify the integrity of the file at install time. If the identical hash cannot be computed from the package file, the package will not be installed.  |

## Example

The following code example defines packages for the .NET Framework redistributable package and its dependencies, such as the Windows Installer.

```
<PackageFiles>
  <PackageFile Name="instmsia.exe" HomeSite="InstMsiAExe"
  PublicKey="3082010A0282010100AA99BD39A81827F42B3D0B4C3F7C772EA7CBB5D18C0DC23A74D793B5E0A04B3F595ECE454F9A792
  9F149CC1A47EE55C2083E1220F855F2EE5FD3E0CA96BC30DEFE58C82732D08554E8F09110BBF32BBE19E5039B0B861DF3B0398CB8FD0
  B1D3C7326AC572BCA29A215908215E277A34052038B9DC270BA1FE934F6F335924E5583F8DA30B620DE5706B55A4206DE59CBF2DFA6B
  D154771192523D2CB6F9B1979DF6A5BF176057929FCC356CA8F440885558ACBC80F464B55CB8C96774A87E8A94106C7FF0DE96857637
  2C36957B443CF323A30DC1BE9D543262A79FE95DB226724C92FD034E3E6FB514986B83CD0255FD6EC9E036187A96840C7F8E203E6CF0
  50203010001"/>
  <PackageFile Name="WindowsInstaller-KB884016-v2-x86.exe" HomeSite="Msi30Exe"
  PublicKey="3082010A0282010100B22D8709B55CDF5599EB5262E7D3F4E34571A932BF94F20EE90DADFE9DC7046A584E9CA4D1D8444
  1FB647E0F65EEC817DA4DDBD9D650B40C565B6C16884BBF03EE504883EC4F88939A51E394197FFAB397A5CE606D9FDD4C9338BDCD345
  971E686CEE98399A096B8EAE0445B1342B93A484E5472F70896E400C482017643AF61C2DBFAE5C5F00213DDF835B40F0D5236467443B
  1A2CA9CDD7E99F1351177FB1526018ECFE0B804782A15FD72C66076910CE74FB218181B6989B4F12F211B66EACA91C7460DB91758715
  856866523D10232AE64A06FDA5295FDFBDD8D34F5C10C35A347D7E91B6AFA0F45B4E8321D7019BDD1F9E5641FEB8737EA6FD40D838FF
  D0203010001"/>
  <PackageFile Name="dotnetfx.exe" HomeSite="DotNetFXExe"
  PublicKey="3082010A0282010100B22D8709B55CDF5599EB5262E7D3F4E34571A932BF94F20EE90DADFE9DC7046A584E9CA4D1D8444
  1FB647E0F65EEC817DA4DDBD9D650B40C565B6C16884BBF03EE504883EC4F88939A51E394197FFAB397A5CE606D9FDD4C9338BDCD345
  971E686CEE98399A096B8EAE0445B1342B93A484E5472F70896E400C482017643AF61C2DBFAE5C5F00213DDF835B40F0D5236467443B
  1A2CA9CDD7E99F1351177FB1526018ECFE0B804782A15FD72C66076910CE74FB218181B6989B4F12F211B66EACA91C7460DB91758715
  856866523D10232AE64A06FDA5295FDFBDD8D34F5C10C35A347D7E91B6AFA0F45B4E8321D7019BDD1F9E5641FEB8737EA6FD40D838FF
  D0203010001"/>
  <PackageFile Name="dotnetchk.exe"/>
</PackageFiles>
```

## See also

- [<Product> element](#)
- [<Package> element](#)

- [Product and package schema reference](#)

# <Strings> element (bootstrapper)

3/5/2021 • 2 minutes to read • [Edit Online](#)

Defines localized strings for product names, package names, and installation error messages.

## Syntax

```
<Strings>
  <String
    Name
  >
</String>
</Strings>
```

## Elements and attributes

The `Strings` element is a child of the `Package` element. It has no attributes.

## String

The `String` element is a child of the `Strings` element. A `Strings` element may have one or more `String` elements.

`String` has the following attribute.

| ATTRIBUTE         | DESCRIPTION                       |
|-------------------|-----------------------------------|
| <code>Name</code> | Required. The name of the string. |

## Example

The following code example specifies all of the English strings for the .NET Framework installer.

```
<Strings>
  <String Name="DisplayName">.NET Framework 2.0</String>
  <String Name="Culture">en</String>
  <String Name="AdminRequired">Administrator permissions are required to install the .NET Framework 2.0.
Contact your administrator.</String>
  <String Name="InvalidPlatformWin9x">Installation of the .NET Framework 2.0 is not supported on Windows
95. Contact your application vendor.</String>
  <String Name="InvalidPlatformWinNT">Installation of the .NET Framework 2.0 is not supported on Windows
NT 4.0. Contact your application vendor.</String>
  <String Name="InvalidPlatformIE">Installation of the .NET Framework 2.0 requires Internet Explorer 5.01
or greater. Contact your application vendor.</String>
  <String Name="InvalidPlatformArchitecture">This version of the .NET Framework 2.0 is not supported on a
64-bit operating system. Contact your application vendor.</String>
  <String Name="WindowsInstallerImproperInstall">Due to an error with Windows Installer, the installation
of the .NET Framework 2.0 cannot proceed.</String>
  <String Name="AnotherInstanceRunning">Another instance of setup is already running. The running instance
must complete before this setup can proceed.</String>
  <String Name="BetaNDPFailure">A beta version of the .NET Framework was detected on the computer.
Uninstall any previous beta versions of .NET Framework before continuing.</String>
  <String Name="GeneralFailure">A failure occurred attempting to install the .NET Framework 2.0.</String>
  <String Name="DotNetFXExe">http://go.microsoft.com/fwlink/?LinkId=37283</String>
  <String Name="InstMsiAExe">http://go.microsoft.com/fwlink/?LinkId=37285</String>
  <String Name="Msi30Exe">http://go.microsoft.com/fwlink/?LinkId=37287</String>
</Strings>
```

## See also

- [<Package> element](#)



# <Schedules> element (bootstrapper)

3/5/2021 • 2 minutes to read • [Edit Online](#)

The `Schedules` element contains `Schedule` elements, which define specific times at which commands defined by the `Command` element should be run.

## Syntax

```
<Schedules>
  <Schedule
    Name
  >
    <BuildList />
    <BeforePackage />
    <AfterPackage />
  </Schedule>
</Schedules>
```

## Elements and attributes

The `Schedules` element is a child of the `Product` element. Each `Product` element might have at most one `Schedules` element. The `Schedules` element has no attributes.

## Schedule

The `Schedule` element is a child of the `Schedules` element. A `Schedules` element must have at least one `Schedule` element.

`Schedule` has the following attribute.

| ATTRIBUTE         | DESCRIPTION   |
|-------------------|---|
| <code>Name</code> | Required. The name of the schedule item. This corresponds to the <code>ScheduleName</code> property of the <code>Command</code> element. When a <code>Command</code> references the named schedule, it will only be executed at the time indicated by that <code>Schedule</code> element. Schedules may also be associated with the <code>FailIf</code> and <code>BypassIf</code> elements, which restrict these conditional tests to executing on the specified schedule. For more information, see <a href="#">&lt;Commands&gt; Element</a> . |

A given `Schedule` element may have exactly one of the following children.

## BuildList

The `BuildList` element instructs the installer to execute a command immediately after the bootstrapping application is started.

## BeforePackage

The `BeforePackage` element instructs the installer to execute a command before the specified package is

installed.

## AfterPackage

The `AfterPackage` element instructs the installer to execute a command after the specified package is installed.

## See also

- [<Product> element](#)
- [Product and package schema reference](#)

# ClickOnce reference

3/5/2021 • 2 minutes to read • [Edit Online](#)

The following pages describe the structure of the XML files used to represent ClickOnce applications.

## In this section

### [ClickOnce Application Manifest](#)

Lists and describes the elements and attributes that make up an application manifest.

### [ClickOnce Deployment Manifest](#)

Lists and describes the elements and attributes that make up a deployment manifest.

### [Product and Package Schema Reference](#)

Lists product and package file elements.

### [ClickOnce Unmanaged API Reference](#)

Lists unmanaged public APIs from dfshim.dll.

## Reference

## Related sections

### [ClickOnce security and deployment](#)

Provides detailed conceptual information about ClickOnce deployment.

### [System.Deployment.Application](#)

Provides links to reference documentation of the public classes that support ClickOnce within managed code.

### [Publish ClickOnce applications](#)

Provides walkthroughs and how-to's that perform ClickOnce tasks.

# ClickOnce application manifest

3/5/2021 • 2 minutes to read • [Edit Online](#)

A ClickOnce application manifest is an XML file that describes an application that is deployed using ClickOnce.

ClickOnce application manifests have the following elements and attributes.

| ELEMENT  | DESCRIPTION   | ATTRIBUTES   |
|--|---|--|
| <a href="#">&lt;assembly&gt; Element</a>         | Required. Top-level element.  | <a href="#">manifestVersion</a>  |
| <a href="#">&lt;assemblyIdentity&gt; Element</a> | Required. Identifies the primary assembly of the ClickOnce application.   | <a href="#">name</a><br><a href="#">version</a><br><a href="#">publicKeyToken</a><br><a href="#">processorArchitecture</a><br><a href="#">language</a> |
| <a href="#">&lt;trustInfo&gt; Element</a>        | Identifies the application security requirements.   | None   |
| <a href="#">&lt;entryPoint&gt; Element</a>       | Required. Identifies the application code entry point.  | <a href="#">name</a>   |
| <a href="#">&lt;dependency&gt; Element</a>       | Required. Identifies each dependency required for the application to run. Optionally identifies assemblies that need to be preinstalled.                      | None   |
| <a href="#">&lt;file&gt; Element</a>             | Optional. Identifies each nonassembly file that is used by the application. Can include Component Object Model (COM) isolation data associated with the file. | <a href="#">name</a><br><a href="#">size</a><br><a href="#">group</a><br><a href="#">optional</a><br><a href="#">writeableType</a>                     |
| <a href="#">&lt;fileAssociation&gt; Element</a>  | Optional. Identifies a file extension to be associated with the application.  | <a href="#">extension</a><br><a href="#">description</a><br><a href="#">progid</a><br><a href="#">defaultIcon</a>                                      |

## Remarks

The ClickOnce application manifest file identifies an application deployed using ClickOnce. For more information

about ClickOnce, see [ClickOnce Security and Deployment](#).

## File location

A ClickOnce application manifest is specific to a single version of a deployment. For this reason, they should be stored separately from deployment manifests. The common convention is to place them in a subdirectory named after the associated version.

The application manifest always must be signed prior to deployment. If you change an application manifest manually, you must use *mage.exe* to re-sign the application manifest, update the deployment manifest, and then re-sign the deployment manifest. For more information, see [Walkthrough: Manually deploy a ClickOnce application](#).

## File name syntax

The name of a ClickOnce application manifest file should be the full name and extension of the application as identified in the `assemblyIdentity` element, followed by the extension *.manifest*. For example, an application manifest that refers to the *Example.exe* application would use the following file name syntax.

```
example.exe.manifest
```

## Example

The following code example shows an application manifest for a ClickOnce application.

```
<?xml version="1.0" encoding="utf-8"?>
<asmv1:assembly xsi:schemaLocation="urn:schemas-microsoft-com:asm.v1 assembly.adaptive.xsd"
manifestVersion="1.0" xmlns:asmv3="urn:schemas-microsoft-com:asm.v3"
xmlns:dsig="http://www.w3.org/2000/09/xmldsig#" xmlns:co.v2="urn:schemas-microsoft-com:clickonce.v2"
xmlns="urn:schemas-microsoft-com:asm.v2" xmlns:asmv1="urn:schemas-microsoft-com:asm.v1"
xmlns:asmv2="urn:schemas-microsoft-com:asm.v2" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:co.v1="urn:schemas-microsoft-com:clickonce.v1">
  <asmv1:assemblyIdentity name="My Application Deployment.exe" version="1.0.0.0"
publicKeyToken="43cb1e8e7a352766" language="neutral" processorArchitecture="x86" type="win32" />
  <application />
  <entryPoint>
    <assemblyIdentity name="MyApplication" version="1.0.0.0" language="neutral" processorArchitecture="x86"
/>
    <commandLine file="MyApplication.exe" parameters="" />
  </entryPoint>
  <trustInfo>
    <security>
      <applicationRequestMinimum>
        <PermissionSet Unrestricted="true" ID="Custom" SameSite="site" />
        <defaultAssemblyRequest permissionSetReference="Custom" />
      </applicationRequestMinimum>
      <requestedPrivileges xmlns="urn:schemas-microsoft-com:asm.v3">
        <!--
          UAC Manifest Options
          If you want to change the Windows User Account Control level replace the
          requestedExecutionLevel node with one of the following.

          <requestedExecutionLevel level="asInvoker" uiAccess="false" />
          <requestedExecutionLevel level="requireAdministrator" uiAccess="false" />
          <requestedExecutionLevel level="highestAvailable" uiAccess="false" />

          If you want to utilize File and Registry Virtualization for backward
          compatibility then delete the requestedExecutionLevel node.
        -->
        <requestedExecutionLevel level="asInvoker" uiAccess="false" />
      </requestedPrivileges>
    </security>
  </trustInfo>
</asmv1:assembly>
```

```

</trustInfo>
<dependency>
  <dependentOS>
    <osVersionInfo>
      <os majorVersion="4" minorVersion="10" buildNumber="0" servicePackMajor="0" />
    </osVersionInfo>
  </dependentOS>
</dependency>
<dependency>
  <dependentAssembly dependencyType="preRequisite" allowDelayedBinding="true">
    <assemblyIdentity name="Microsoft.Windows.CommonLanguageRuntime" version="4.0.20506.0" />
  </dependentAssembly>
</dependency>
<dependency>
  <dependentAssembly dependencyType="install" allowDelayedBinding="true" codebase="MyApplication.exe"
size="4096">
    <assemblyIdentity name="MyApplication" version="1.0.0.0" language="neutral"
processorArchitecture="x86" />
    <hash>
      <dsig:Transforms>
        <dsig:Transform Algorithm="urn:schemas-microsoft-com:HashTransforms.Identity" />
      </dsig:Transforms>
      <dsig:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
      <dsig:DigestValue>DpTW7RzS9IeT/RBSLj54vfTEzNg=</dsig:DigestValue>
    </hash>
  </dependentAssembly>
</dependency>
<publisherIdentity name="CN=DOMAINCONTROLLER\UserMe"
issuerKeyHash="18312a18a21b215ecf4cdb20f5a0e0b0dd263c08" /><Signature Id="StrongNameSignature"
xmlns="http://www.w3.org/2000/09/xmldsig#">
...
</Signature></r:issuer></r:license></msrel:RelData></KeyInfo></Signature></asmv1:assembly>

```

## See also

- [Publish ClickOnce applications](#)

# <assembly> Element (ClickOnce Application)

3/5/2021 • 2 minutes to read • [Edit Online](#)

The top-level element for the application manifest.

## Syntax

```
<assembly  
  manifestVersion  
>
```

## Elements and attributes

The `assembly` element is the root element and is required. Its first contained element must be an `assemblyIdentity` element. The manifest elements must be in one of the following namespaces:

`urn:schemas-microsoft-com:asm.v1`

`urn:schemas-microsoft-com:asm.v2`

`http://www.w3.org/2000/09/xmldsig#`

Child elements of the assembly must also be in these namespaces, by inheritance or by tagging.

The `assembly` element has the following attribute.

| ATTRIBUTE                    | DESCRIPTION  |
|------------------------------|--|
| <code>manifestVersion</code> | Required. The <code>manifestVersion</code> attribute must be set to <code>1.0</code> . |

## Example

The following code example illustrates an `assembly` element in an application manifest for a ClickOnce application. This code example is part of a larger example provided in [ClickOnce application manifest](#).

```
<asmv1:assembly  
  xsi:schemaLocation="urn:schemas-microsoft-com:asm.v1 assembly.adaptive.xsd"  
  manifestVersion="1.0"  
  xmlns:asmv3="urn:schemas-microsoft-com:asm.v3"  
  xmlns:dsig=http://www.w3.org/2000/09/xmldsig#  
  xmlns:co.v2="urn:schemas-microsoft-com:clickonce.v2"  
  xmlns="urn:schemas-microsoft-com:asm.v2"  
  xmlns:asmv1="urn:schemas-microsoft-com:asm.v1"  
  xmlns:asmv2="urn:schemas-microsoft-com:asm.v2"  
  xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance  
  xmlns:co.v1="urn:schemas-microsoft-com:clickonce.v1">
```

## See also

- [ClickOnce application manifest](#)

- `<assembly>` element



# <assemblyIdentity> element (ClickOnce application)

3/5/2021 • 2 minutes to read • [Edit Online](#)

Identifies the application deployed in a ClickOnce deployment.

## Syntax

```
<assemblyIdentity
  name
  version
  publicKeyToken
  processorArchitecture
  language
/>
```

## Elements and attributes

The `assemblyIdentity` element is required. It contains no child elements and has the following attributes.

| ATTRIBUTE                          | DESCRIPTION   |
|------------------------------------|---|
| <code>Name</code>                  | <p>Required. Identifies the name of the application.</p> <p>If <code>Name</code> contains special characters, such as single or double quotes, the application may fail to activate.</p>  |
| <code>Version</code>               | <p>Required. Specifies the version number of the application in the following format: <code>major.minor.build.revision</code></p>   |
| <code>publicKeyToken</code>        | <p>Optional. Specifies a 16-character hexadecimal string that represents the last 8 bytes of the <code>SHA-1</code> hash value of the public key under which the application or assembly is signed. The public key that is used to sign the catalog must be 2048 bits or greater.</p> <p>Although signing an assembly is recommended but optional, this attribute is required. If an assembly is unsigned, you should copy a value from a self-signed assembly or use a "dummy" value of all zeros.</p> |
| <code>processorArchitecture</code> | <p>Required. Specifies the processor. The valid values are <code>msil</code> for all processors, <code>x86</code> for 32-bit Windows, <code>IA64</code> for 64-bit Windows, and <code>Itanium</code> for Intel 64-bit Itanium processors.</p>   |
| <code>language</code>              | <p>Required. Identifies the two part language codes (for example, <code>en-US</code>) of the assembly. This element is in the <code>asmv2</code> namespace. If unspecified, the default is <code>neutral</code>.</p>  |

## Examples

## Description

The following code example illustrates an `assemblyIdentity` element in a ClickOnce application manifest. This code example is part of a larger example provided in [ClickOnce Application Manifest](#).

## Code

```
<asmv1:assemblyIdentity
  name="My Application Deployment.exe"
  version="1.0.0.0"
  publicKeyToken="43cb1e8e7a352766"
  language="neutral"
  processorArchitecture="x86"
  type="win32" />
```

## See also

- [ClickOnce application manifest](#)
- [<assemblyIdentity> element](#)

# <trustInfo> element (ClickOnce application)

3/5/2021 • 4 minutes to read • [Edit Online](#)

Describes the minimum security permissions required for the application to run on the client computer.

## Syntax

```
<trustInfo>
<security>
  <applicationRequestMinimum>
    <PermissionSet
      ID
      Unrestricted>
    <IPermission
      class
      version
      Unrestricted
    />
  </PermissionSet>
  <defaultAssemblyRequest
    permissionSetReference
  />
  <assemblyRequest
    name
    permissionSetReference
  />
</applicationRequestMinimum>
<requestedPrivileges>
  <requestedExecutionLevel
    level
    uiAccess
  />
</requestedPrivileges>
</security>
</trustInfo>
```

## Elements and attributes

The `trustInfo` element is required and is in the `asm.v2` namespace. It has no attributes and contains the following elements.

### security

Required. This element is a child of the `trustInfo` element. It contains the `applicationRequestMinimum` element and has no attributes.

### applicationRequestMinimum

Required. This element is a child of the `security` element and contains the `PermissionSet`, `assemblyRequest`, and `defaultAssemblyRequest` elements. This element has no attributes.

### PermissionSet

Required. This element is a child of the `applicationRequestMinimum` element and contains the `IPermission` element. This element has the following attributes.

- `ID`

Required. Identifies the permission set. This attribute can be any value. The ID is referenced in the `defaultAssemblyRequest` and `assemblyRequest` attributes.

- `version`

Required. Identifies the version of the permission. Normally this value is `1`.

## IPermission

Optional. This element is a child of the `PermissionSet` element. The `IPermission` element fully identifies a permission class in the .NET Framework. The `IPermission` element has the following attributes, but can have additional attributes that correspond to properties on the permission class. To find out the syntax for a specific permission, see the examples listed in the Security.config file.

- `class`

Required. Identifies the permission class by strong name. For example, the following code identifies the `FileDialogPermission` type.

```
System.Security.Permissions.FileDialogPermission, mscorlib, Version=1.2.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089
```

- `version`

Required. Identifies the version of the permission. Usually this value is `1`.

- `Unrestricted`

Required. Identifies whether the application needs an unrestricted grant of this permission. If `true`, the permission grant is unconditional. If `false`, or if this attribute is undefined, it is restricted according to the permission-specific attributes defined on the `IPermission` tag. Take the following permissions:

```
<IPermission
  class="System.Security.Permissions.EnvironmentPermission, mscorlib, Version=1.2.3300.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089"
  version="1"
  Read="USERNAME" />
<IPermission
  class="System.Security.Permissions.FileDialogPermission, mscorlib, Version=1.2.3300.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089"
  version="1"
  Unrestricted="true" />
```

In this example, the declaration for `EnvironmentPermission` restricts the application to reading only the environment variable USERNAME, whereas the declaration for `FileDialogPermission` gives the application unrestricted use of all `FileDialog` classes.

## defaultAssemblyRequest

Optional. Identifies the set of permissions granted to all assemblies. This element is a child of the `applicationRequestMinimum` element and has the following attribute.

- `permissionSetReference`

Required. Identifies the ID of the permission set that is the default permission. The permission set is

declared in the `PermissionSet` element.

## assemblyRequest

Optional. Identifies permissions for a specific assembly. This element is a child of the `applicationRequestMinimum` element and has the following attributes.

- `Name`

Required. Identifies the assembly name.

- `permissionSetReference`

Required. Identifies the ID of the permission set that this assembly requires. The permission set is declared in the `PermissionSet` element.

## requestedPrivileges

Optional. This element is a child of the `security` element and contains the `requestedExecutionLevel` element. This element has no attributes.

## requestedExecutionLevel

Optional. Identifies the security level at which the application requests to be executed. This element has no children and has the following attributes.

- `Level`

Required. Indicates the security level the application is requesting. Possible values are:

`asInvoker`, requesting no additional permissions. This level requires no additional trust prompts.

`highestAvailable`, requesting the highest permissions available to the parent process.

`requireAdministrator`, requesting full administrator permissions.

ClickOnce applications will only install with a value of `asInvoker`. Installing with any other value will fail.

- `uiAccess`

Optional. Indicates whether the application requires access to protected user interface elements. Values are either `true` or `false`, and the default is false. Only signed applications should have a value of true.

## Remarks

If a ClickOnce application asks for more permissions than the client computer will grant by default, the common language runtime's Trust Manager will ask the user if she wants to grant the application this elevated level of trust. If she says no, the application will not run; otherwise, it will run with the requested permissions.

All permissions requested using `defaultAssemblyRequest` and `assemblyRequest` will be granted without user prompting if the deployment manifest has a valid Trust License.

For more information about Permission Elevation, see [Securing ClickOnce Applications](#). For more information about policy deployment, see [Trusted Application Deployment Overview](#).

## Examples

The following three code examples illustrate `trustInfo` elements for the default named security zones—

Internet, LocalIntranet, and FullTrust—for use in a ClickOnce deployment's application manifest.

The first example illustrates the `trustInfo` element for the default permissions available in the Internet security zone.

```
<trustInfo>
  <security>
    <applicationRequestMinimum>
      <PermissionSet ID="Internet">
        <IPermission
          class="System.Security.Permissions.FileDialogPermission, mscorlib, Version=1.2.3300.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089"
          version="1"
          Access="Open" />
        <IPermission
          class="System.Security.Permissions.IsolatedStorageFilePermission, mscorlib, Version=1.2.3300.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089"
          version="1"
          Allowed="DomainIsolationByUser"
          UserQuota="10240" />
        <IPermission
          class="System.Security.Permissions.SecurityPermission, mscorlib, Version=1.2.3300.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089"
          version="1"
          Flags="Execution" />
        <IPermission
          class="System.Security.Permissions.UIPermission, mscorlib, Version=1.2.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089"
          version="1"
          Window="SafeTopLevelWindows"
          Clipboard="OwnClipboard" />
        <IPermission
          class="System.Drawing.Printing.PrintingPermission, System.Drawing, Version=1.2.3300.0,
Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
          version="1"
          Level="SafePrinting" />
      </PermissionSet>
      <defaultAssemblyRequest permissionSetReference="Internet" />
    </applicationRequestMinimum>
  </security>
</trustInfo>
```

The second example illustrates the `trustInfo` element for the default permissions available in the LocalIntranet security zone.

```

<trustInfo>
  <security>
    <applicationRequestMinimum>
      <PermissionSet ID="LocalIntranet">
        <IPermission
          class="System.Security.Permissions.EnvironmentPermission, mscorlib, Version=1.2.3300.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089"
          version="1"
          Read="USERNAME" />
        <IPermission
          class="System.Security.Permissions.FileDialogPermission, mscorlib, Version=1.2.3300.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089"
          version="1"
          Unrestricted="true" />
        <IPermission
          class="System.Security.Permissions.IsolatedStorageFilePermission, mscorlib, Version=1.2.3300.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089"
          version="1"
          Allowed="AssemblyIsolationByUser"
          UserQuota="9223372036854775807"
          Expiry="9223372036854775807"
          Permanent="True" />
        <IPermission
          class="System.Security.Permissions.ReflectionPermission, mscorlib, Version=1.2.3300.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089"
          version="1"
          Flags="ReflectionEmit" />
        <IPermission
          class="System.Security.Permissions.SecurityPermission, mscorlib, Version=1.2.3300.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089"
          version="1"
          Flags="Assertion, Execution" />
        <IPermission
          class="System.Security.Permissions.UIPermission, mscorlib, Version=1.2.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089"
          version="1"
          Unrestricted="true" />
        <IPermission
          class="System.Net.DnsPermission, System, Version=1.2.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089"
          version="1"
          Unrestricted="true" />
        <IPermission
          class="System.Drawing.Printing.PrintingPermission, System.Drawing, Version=1.2.3300.0,
Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
          version="1"
          Level="DefaultPrinting" />
        <IPermission
          class="System.Diagnostics.EventLogPermission, System, Version=1.2.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089"
          version="1" />
      </PermissionSet>
      <defaultAssemblyRequest permissionSetReference="LocalIntranet" />
    </applicationRequestMinimum>
  </security>
</trustInfo>

```

The third example illustrates the `trustInfo` element for the default permissions available in the FullTrust security zone.

```
<trustInfo>
  <security>
    <applicationRequestMinimum>
      <PermissionSet ID="FullTrust" Unrestricted="true" />
      <defaultAssemblyRequest permissionSetReference="FullTrust" />
    </applicationRequestMinimum>
  </security>
</trustInfo>
```

## See also

- [Trusted Application Deployment overview](#)
- [ClickOnce application manifest](#)



# <entryPoint> element (ClickOnce application)

3/5/2021 • 2 minutes to read • [Edit Online](#)

Identifies the assembly that should be executed when this ClickOnce application is run on a client computer.

## Syntax

```
<entryPoint
  name
>
  <assemblyIdentity
    name
    version
    processorArchitecture
    language
  />
  <commandLine
    file
    parameters
  />
  <customHostRequired />
  <customUX />
</entryPoint>
```

## Elements and attributes

The `entryPoint` element is required and is in the `urn:schemas-microsoft-com:asm.v2` namespace. There may only be one `entryPoint` element defined in an application manifest.

The `entryPoint` element has the following attribute.

| ATTRIBUTE         | DESCRIPTION   |
|-------------------|---|
| <code>name</code> | Optional. This value is not used by .NET Framework. |

`entryPoint` has the following elements.

## assemblyIdentity

Required. The role of `assemblyIdentity` and its attributes is defined in [<assemblyIdentity> Element](#).

The `processorArchitecture` attribute of this element and the `processorArchitecture` attribute defined in the `assemblyIdentity` elsewhere in the application manifest must match.

## commandLine

Required. Must be a child of the `entryPoint` element. It has no child elements and has the following attributes.

| ATTRIBUTE | DESCRIPTION |
|-----------|-------------|
|-----------|-------------|

| ATTRIBUTE               | DESCRIPTION   |
|-------------------------|---|
| <code>file</code>       | Required. A local reference to the startup assembly for the ClickOnce application. This value cannot contain forward slash (/) or backslash (\) path separators.    |
| <code>parameters</code> | Required. Describes the action to take with the entry point. The only valid value is <code>run</code> ; if a blank string is supplied, <code>run</code> is assumed. |

## customHostRequired

Optional. If included, specifies that this deployment contains a component that will be deployed inside of a custom host, and is not a stand-alone application.

If this element is present, the `assemblyIdentity` and `commandLine` elements must not also be present. If they are, ClickOnce will raise a validation error during installation.

This element has no attributes and no children.

## customUX

Optional. Specifies that the application is installed and maintained by a custom installer, and does not create a Start menu entry, shortcut, or Add or Remove Programs entry.

```
<customUX xmlns="urn:schemas-microsoft-com:clickonce.v1" />
```

An application that includes the customUX element must provide a custom installer that uses the [InPlaceHostingManager](#) class to perform install operations. An application with this element cannot be installed by double-clicking its manifest or setup.exe prerequisite bootstrapper. The custom installer can create Start menu entries, shortcuts, and Add or Remove Programs entries. If the custom installer does not create an Add or Remove Programs entry, it must store the subscription identifier provided by the [SubscriptionIdentity](#) property and enable the user to uninstall the application later by calling the [UninstallCustomUXApplication](#) method. For more information, see [Walkthrough: Creating a Custom Installer for a ClickOnce Application](#).

## Remarks

This element identifies the assembly and entry point for the ClickOnce application.

You cannot use `commandLine` to pass parameters into your application at run time. You can access query string parameters for a ClickOnce deployment from the application's [AppDomain](#). For more information, see [How to: Retrieve Query String Information in an Online ClickOnce Application](#).

## Example

The following code example illustrates an `entryPoint` element in an application manifest for a ClickOnce application. This code example is part of a larger example provided for the [ClickOnce Application Manifest](#) topic.

```
<!-- Identify the main code entrypoint. -->
<!-- This code runs the main method in an executable assembly. -->
<entryPoint>
  <assemblyIdentity
    name="MyApplication"
    version="1.0.0.0"
    language="neutral"
    processorArchitecture="x86" />
  <commandLine file="MyApplication.exe" parameters="" />
</entryPoint>
```

## See also

- [ClickOnce application manifest](#)

# <dependency> element (ClickOnce application)

3/5/2021 • 4 minutes to read • [Edit Online](#)

Identifies a platform or assembly dependency that is required for the application.

## Syntax

```
<dependency>
  <dependentOS
    supportURL
    description
  >
    <osVersionInfo>
      <os
        majorVersion
        minorVersion
        buildNumber
        servicePackMajor
        servicePackMinor
        productType
        suiteType
      />
    </osVersionInfo>
  </dependentOS>
  <dependentAssembly
    dependencyType
    allowDelayedBinding
    group
    codeBase
    size
  >
    <assemblyIdentity
      name
      version
      processorArchitecture
      language
    >
      <hash>
        <dsig:Transforms>
          <dsig:Transform
            Algorithm
          />
        </dsig:Transforms>
        <dsig:DigestMethod />
        <dsig:DigestValue>
        </dsig:DigestValue>
      </hash>
    </assemblyIdentity>
  </dependentAssembly>
</dependency>
```

## Elements and attributes

The `dependency` element is required. There may be multiple instances of `dependency` in the same application manifest.

The `dependency` element has no attributes, and contains the following child elements.

## dependentOS

Optional. Contains the `osVersionInfo` element. The `dependentOS` and `dependentAssembly` elements are mutually exclusive: one or the other must exist for a `dependency` element, but not both.

`dependentOS` supports the following attributes.

| ATTRIBUTE                | DESCRIPTION  |
|--------------------------|--|
| <code>supportUrl</code>  | Optional. Specifies a support URL for the dependent platform. This URL is shown to the user if the required platform is found. |
| <code>description</code> | Optional. Describes, in human-readable form, the operating system described by the <code>dependentOS</code> element.           |

## osVersionInfo

Required. This element is a child of the `dependentOS` element and contains the `os` element. This element has no attributes.

## os

Required. This element is a child of the `osVersionInfo` element. This element has the following attributes.

| ATTRIBUTE                     | DESCRIPTION   |
|-------------------------------|---|
| <code>majorVersion</code>     | Required. Specifies the major version number of the OS.   |
| <code>minorVersion</code>     | Required. Specifies the minor version number of the OS.   |
| <code>buildNumber</code>      | Required. Specifies the build number of the OS.   |
| <code>servicePackMajor</code> | Required. Specifies the service pack major number of the OS.  |
| <code>servicePackMinor</code> | Optional. Specifies the service pack minor number of the OS.  |
| <code>productType</code>      | Optional. Identifies the product type value. Valid values are <code>server</code> , <code>workstation</code> , and <code>domainController</code> . For example, for Windows 2000 Professional, this attribute value is <code>workstation</code> .   |
| <code>suiteType</code>        | Optional. Identifies a product suite available on the system, or the system's configuration type. Valid values are <code>backoffice</code> , <code>blade</code> , <code>datacenter</code> , <code>enterprise</code> , <code>home</code> , <code>professional</code> , <code>smallbusiness</code> , <code>smallbusinessRestricted</code> , and <code>terminal</code> . For example, for Windows 2000 Professional, this attribute value is <code>professional</code> . |

## dependentAssembly

Optional. Contains the `assemblyIdentity` element. The `dependentOS` and `dependentAssembly` elements are mutually exclusive: one or the other must exist for a `dependency` element, but not both.

`dependentAssembly` has the following attributes.

| ATTRIBUTE                        | DESCRIPTION  |
|----------------------------------|--|
| <code>dependencyType</code>      | Required. Specifies the dependency type. Valid values are <code>prerequisite</code> and <code>install</code> . An <code>install</code> assembly is installed as part of the ClickOnce application. A <code>prerequisite</code> assembly must be present in the global assembly cache (GAC) before the ClickOnce application can install.   |
| <code>allowDelayedBinding</code> | Required. Specifies whether the assembly can be loaded programmatically at run time.   |
| <code>group</code>               | Optional. If the <code>dependencyType</code> attribute is set to <code>install</code> , designates a named group of assemblies that only install on demand. For more information, see <a href="#">Walkthrough: Downloading Assemblies on Demand with the ClickOnce Deployment API Using the Designer</a> .<br><br>If set to <code>framework</code> and the <code>dependencyType</code> attribute is set to <code>prerequisite</code> , designates the assembly as part of the .NET Framework. The global assembly cache (GAC) is not checked for this assembly when installing on .NET Framework 4 and later versions. |
| <code>codeBase</code>            | Required when the <code>dependencyType</code> attribute is set to <code>install</code> . The path to the dependent assembly. May be either an absolute path, or a path relative to the manifest's code base. This path must be a valid URI in order for the assembly manifest to be valid.   |
| <code>size</code>                | Required when the <code>dependencyType</code> attribute is set to <code>install</code> . The size of the dependent assembly, in bytes.   |

## assemblyIdentity

Required. This element is a child of the `dependentAssembly` element and has the following attributes.

| ATTRIBUTE                          | DESCRIPTION   |
|------------------------------------|---|
| <code>name</code>                  | Required. Identifies the name of the application.   |
| <code>version</code>               | Required. Specifies the version number of the application in the following format: <code>major.minor.build.revision</code>  |
| <code>publicKeyToken</code>        | Optional. Specifies a 16-character hexadecimal string that represents the last 8 bytes of the <code>SHA-1</code> hash value of the public key under which the application or assembly is signed. The public key used to sign the catalog must be 2048 bits or more. |
| <code>processorArchitecture</code> | Optional. Specifies the processor. The valid values are <code>x86</code> for 32-bit Windows and <code>I64</code> for 64-bit Windows.  |
| <code>language</code>              | Optional. Identifies the two part language codes, such as EN-US, of the assembly.   |

## hash

The `hash` element is an optional child of the `assemblyIdentity` element. The `hash` element has no attributes.

ClickOnce uses an algorithmic hash of all the files in an application as a security check, to ensure that none of the files were changed after deployment. If the `hash` element is not included, this check will not be performed. Therefore, omitting the `hash` element is not recommended.

### **dsig:Transforms**

The `dsig:Transforms` element is a required child of the `hash` element. The `dsig:Transforms` element has no attributes.

### **dsig:Transform**

The `dsig:Transform` element is a required child of the `dsig:Transforms` element. The `dsig:Transform` element has the following attributes.

| ATTRIBUTE              | DESCRIPTION  |
|------------------------|--|
| <code>Algorithm</code> | The algorithm used to calculate the digest for this file. Currently the only value used by ClickOnce is <code>urn:schemas-microsoft-com:HashTransforms.Identity</code> . |

### **dsig:DigestMethod**

The `dsig:DigestMethod` element is a required child of the `hash` element. The `dsig:DigestMethod` element has the following attributes.

| ATTRIBUTE              | DESCRIPTION   |
|------------------------|---|
| <code>Algorithm</code> | The algorithm used to calculate the digest for this file. Currently the only value used by ClickOnce is <code>http://www.w3.org/2000/09/xml#sha1</code> . |

### **dsig:DigestValue**

The `dsig:DigestValue` element is a required child of the `hash` element. The `dsig:DigestValue` element has no attributes. Its text value is the computed hash for the specified file.

## Remarks

All assemblies used by your application must have a corresponding `dependency` element. Dependent assemblies do not include assemblies that must be preinstalled in the global assembly cache as platform assemblies.

## Example

The following code example illustrates `dependency` elements in a ClickOnce application manifest. This code example is part of a larger example provided for the [ClickOnce Application Manifest](#) topic.

```

<dependency>
  <dependentOS>
    <osVersionInfo>
      <os
        majorVersion="4"
        minorVersion="10"
        buildNumber="0"
        servicePackMajor="0" />
    </osVersionInfo>
  </dependentOS>
</dependency>
<dependency>
  <dependentAssembly>
    dependencyType="preRequisite"
    allowDelayedBinding="true">
      <assemblyIdentity
        name="Microsoft.Windows.CommonLanguageRuntime"
        version="4.0.20506.0" />
    </dependentAssembly>
  </dependency>

<dependency>
  <dependentAssembly>
    dependencyType="install"
    allowDelayedBinding="true"
    codebase="MyApplication.exe"
    size="4096">
      <assemblyIdentity
        name="MyApplication"
        version="1.0.0.0"
        language="neutral"
        processorArchitecture="x86" />
    <hash>
      <dsig:Transforms>
        <dsig:Transform Algorithm="urn:schemas-microsoft-com:HashTransforms.Identity" />
      </dsig:Transforms>
      <dsig:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
      <dsig:DigestValue>DpTW7RzS9IeT/RBSLj54vfTEzNg=</dsig:DigestValue>
    </hash>
  </dependentAssembly>
</dependency>

```

## See also

- [ClickOnce application manifest](#)
- [<dependency> element](#)



# <file> element (ClickOnce application)

3/5/2021 • 7 minutes to read • [Edit Online](#)

Identifies all nonassembly files downloaded and used by the application.

## Syntax

```
<file
  name
  size
  group
  optional
  writeableType
>
  <typelib
    tlbid
    version
    helpdir
    resourceid
    flags
  />
  <comClass
    clsid
    description
    threadingModel
    tlbid
    progid
    miscStatus
    miscStatusIcon
    miscStatusContent
    miscStatusDocPrint
    miscStatusThumbnail
  />
  <comInterfaceExternalProxyStub
    iid
    baseInterface
    numMethods
    name
    tlbid
    proxyStubClass32
  />
  <comInterfaceProxyStub
    iid
    baseInterface
    numMethods
    name
    tlbid
    proxyStubClass32
  />
  <windowClass
    versioned
  />
</file>
```

## Elements and attributes

The `file` element is optional. The element has the following attributes.

| ATTRIBUTE                 | DESCRIPTION  |
|---------------------------|--|
| <code>name</code>         | Required. Identifies the name of the file.   |
| <code>size</code>         | Required. Specifies the size, in bytes, of the file.   |
| <code>group</code>        | Optional, if the <code>optional</code> attribute is not specified or set to <code>false</code> ; required if <code>optional</code> is <code>true</code> . The name of the group to which this file belongs. The name can be any Unicode string value chosen by the developer, and is used for downloading files on demand with the <a href="#">ApplicationDeployment</a> class.  |
| <code>optional</code>     | Optional. Specifies whether this file must download when the application is first run, or whether the file should reside only on the server until the application requests it on demand. If <code>false</code> or undefined, the file is downloaded when the application is first run or installed. If <code>true</code> , a <code>group</code> must be specified for the application manifest to be valid. <code>optional</code> cannot be true if <code>writableType</code> is specified with the value <code>applicationData</code> . |
| <code>writableType</code> | Optional. Specifies that this file is a data file. Currently the only valid value is <code>applicationData</code> .  |

## typelib

The `typelib` element is an optional child of the `file` element. The element describes the type library that belongs to the COM component. The element has the following attributes.

| ATTRIBUTE               | DESCRIPTION  |
|-------------------------|--|
| <code>tlbid</code>      | Required. The GUID assigned to the type library.   |
| <code>version</code>    | Required. The version number of the type library.  |
| <code>helpdir</code>    | Required. The directory that contains the Help files for the component. May be zero-length.  |
| <code>resourceid</code> | Optional. The hexadecimal string representation of the locale identifier (LCID). It is one to four hexadecimal digits without a 0x prefix and without leading zeros. The LCID may have a neutral sublanguage identifier. |
| <code>flags</code>      | Optional. The string representation of the type library flags for this type library. Specifically, it should be one of "RESTRICTED", "CONTROL", "HIDDEN" and "HASDISKIMAGE".   |

## comClass

The `comClass` element is an optional child of the `file` element, but is required if the ClickOnce application contains a COM component it intends to deploy using registration-free COM. The element has the following attributes.

| ATTRIBUTE                       | DESCRIPTION  |
|---------------------------------|--|
| <code>clsid</code>              | Required. The class ID of the COM component expressed as a GUID.   |
| <code>description</code>        | Optional. The class name.  |
| <code>threadingModel</code>     | <p>Optional. The threading model used by in-process COM classes. If this property is null, no threading model is used. The component is created on the main thread of the client and calls from other threads are marshaled to this thread. The following list shows the valid values:</p> <p><code>Apartment</code> , <code>Free</code> , <code>Both</code> , and <code>Neutral</code> .</p>  |
| <code>tlbid</code>              | Optional. GUID for the type library for this COM component.  |
| <code>progid</code>             | <p>Optional. Version-dependent programmatic identifier associated with the COM component. The format of a <code>ProgID</code> is <code>&lt;vendor&gt;.&lt;component&gt;.&lt;version&gt;</code> .</p>   |
| <code>miscStatus</code>         | <p>Optional. Duplicates in the assembly manifest the information provided by the <code>MiscStatus</code> registry key. If values for the <code>miscStatusIcon</code> , <code>miscStatusContent</code> , <code>miscStatusDocprint</code> , or <code>miscStatusThumbnail</code> attributes are not found, the corresponding default value listed in <code>miscStatus</code> is used for the missing attributes. The value can be a comma-delimited list of the attribute values from the following table. You can use this attribute if the COM class is an OCX class that requires <code>MiscStatus</code> registry key values.</p> |
| <code>miscStatusIcon</code>     | <p>Optional. Duplicates in the assembly manifest the information provided by DVASPECT_ICON. It can provide an icon of an object. The value can be a comma-delimited list of the attribute values from the following table. You can use this attribute if the COM class is an OCX class that requires <code>Miscstatus</code> registry key values.</p>  |
| <code>miscStatusContent</code>  | <p>Optional. Duplicates in the assembly manifest the information provided by DVASPECT_CONTENT. It can provide a compound document displayable for a screen or printer. The value can be a comma-delimited list of the attribute values from the following table. You can use this attribute if the COM class is an OCX class that requires <code>MiscStatus</code> registry key values.</p>  |
| <code>miscStatusDocPrint</code> | <p>Optional. Duplicates in the assembly manifest the information provided by DVASPECT_DOCPRINT. It can provide an object representation displayable on the screen as if printed to a printer. The value can be a comma-delimited list of the attribute values from the following table. You can use this attribute if the COM class is an OCX class that requires <code>MiscStatus</code> registry key values.</p>   |

| ATTRIBUTE                        | DESCRIPTION   |
|----------------------------------|---|
| <code>miscStatusThumbnail</code> | Optional. Duplicates in an assembly manifest the information provided by DVASPECT_THUMBNAIL. It can provide a thumbnail of an object displayable in a browsing tool. The value can be a comma-delimited list of the attribute values from the following table. You can use this attribute if the COM class is an OCX class that requires <code>MiscStatus</code> registry key values. |

## comInterfaceExternalProxyStub

The `comInterfaceExternalProxyStub` element is an optional child of the `file` element, but may be required if the ClickOnce application contains a COM component it intends to deploy using registration-free COM. The element contains the following attributes.

| ATTRIBUTE                     | DESCRIPTION  |
|-------------------------------|--|
| <code>iid</code>              | Required. The interface ID (IID) which is served by this proxy. The IID must have braces surrounding it.               |
| <code>baseInterface</code>    | Optional. The IID of the interface from which the interface referenced by <code>iid</code> is derived.                 |
| <code>numMethods</code>       | Optional. The number of methods implemented by the interface.  |
| <code>name</code>             | Optional. The name of the interface as it will appear in code.   |
| <code>tlbid</code>            | Optional. The type library that contains the description of the interface specified by the <code>iid</code> attribute. |
| <code>proxyStubClass32</code> | Optional. Maps an IID to a CLSID in 32-bit proxy DLLs.   |

## comInterfaceProxyStub

The `comInterfaceProxyStub` element is an optional child of the `file` element, but may be required if the ClickOnce application contains a COM component it intends to deploy using registration-free COM. The element contains the following attributes.

| ATTRIBUTE                  | DESCRIPTION  |
|----------------------------|--|
| <code>iid</code>           | Required. The interface ID (IID) which is served by this proxy. The IID must have braces surrounding it. |
| <code>baseInterface</code> | Optional. The IID of the interface from which the interface referenced by <code>iid</code> is derived.   |
| <code>numMethods</code>    | Optional. The number of methods implemented by the interface.  |
| <code>Name</code>          | Optional. The name of the interface as it will appear in code.   |

| ATTRIBUTE                     | DESCRIPTION  |
|-------------------------------|--|
| <code>Tlbid</code>            | Optional. The type library that contains the description of the interface specified by the <code>iid</code> attribute.   |
| <code>proxyStubClass32</code> | Optional. Maps an IID to a CLSID in 32-bit proxy DLLs.   |
| <code>threadingModel</code>   | Optional. Optional. The threading model used by in-process COM classes. If this property is null, no threading model is used. The component is created on the main thread of the client and calls from other threads are marshaled to this thread. The following list shows the valid values:<br><br><code>Apartment</code> , <code>Free</code> , <code>Both</code> , and <code>Neutral</code> . |

## windowClass

The `windowClass` element is an optional child of the `file` element, but may be required if the ClickOnce application contains a COM component it intends to deploy using registration-free COM. The element refers to a window class defined by the COM component that must have a version applied to it. The element contains the following attributes.

| ATTRIBUTE              | DESCRIPTION  |
|------------------------|--|
| <code>versioned</code> | Optional. Controls whether the internal window class name used in registration contains the version of the assembly that contains the window class. The value of this attribute can be <code>yes</code> or <code>no</code> . The default is <code>yes</code> . The value <code>no</code> should only be used if the same window class is defined by a side-by-side component and an equivalent non-side-by-side component and you want to treat them as the same window class. Note that the usual rules about window class registration apply—only the first component that registers the window class will be able to register it, because it does not have a version applied to it. |

## hash

The `hash` element is an optional child of the `file` element. The `hash` element has no attributes.

ClickOnce uses an algorithmic hash of all the files in an application as a security check, to ensure that none of the files were changed after deployment. If the `hash` element is not included, this check will not be performed. Therefore, omitting the `hash` element is not recommended.

If a manifest contains a file that is not hashed, that manifest cannot be digitally signed, because users cannot verify the contents of an unhashed file.

## dsig:Transforms

The `dsig:Transforms` element is a required child of the `hash` element. The `dsig:Transforms` element has no attributes.

## dsig:Transform

The `dsig:Transform` element is a required child of the `dsig:Transforms` element. The `dsig:Transform` element has the following attributes.

| ATTRIBUTE | DESCRIPTION  |
|-----------|--|
| Algorithm | The algorithm used to calculate the digest for this file. Currently the only value used by ClickOnce is <code>urn:schemas-microsoft-com:HashTransforms.Identity</code> . |

## dsig:DigestMethod

The `dsig:DigestMethod` element is a required child of the `hash` element. The `dsig:DigestMethod` element has the following attributes.

| ATTRIBUTE | DESCRIPTION  |
|-----------|--|
| Algorithm | The algorithm used to calculate the digest for this file. Currently the only value used by ClickOnce is <code>http://www.w3.org/2000/09/xmlsig#sha1</code> . |

## dsig:DigestValue

The `dsig:DigestValue` element is a required child of the `hash` element. The `dsig:DigestValue` element has no attributes. Its text value is the computed hash for the specified file.

## Remarks

This element identifies all the nonassembly files that make up the application and, in particular, the hash values for file verification. This element can also include Component Object Model (COM) isolation data associated with the file. If a file changes, the application manifest file also must be updated to reflect the change.

## Example

The following code example illustrates `file` elements in an application manifest for an application deployed using ClickOnce.

```
<file name="Icon.ico" size="9216">
  <hash>
    <dsig:Transforms>
      <dsig:Transform Algorithm="urn:schemas-microsoft-com:HashTransforms.Identity" />
    </dsig:Transforms>
    <dsig:DigestMethod Algorithm="http://www.w3.org/2000/09/xmlsig#sha1" />
    <dsig:DigestValue>lVoj+Rh6RQ/HPNLOdayQah5McrI=</dsig:DigestValue>
  </hash>
</file>
```

## See also

- [ClickOnce application manifest](#)

# <fileAssociation> element (ClickOnce application)

3/5/2021 • 2 minutes to read • [Edit Online](#)

Identifies a file extension to be associated with the application.

## Syntax

```
<fileAssociation
  xmlns="urn:schemas-microsoft-com:clickonce.v1"
  extension
  description
  progid
  defaultIcon
/>
```

## Elements and attributes

The `fileAssociation` element is optional. The element has the following attributes.

| ATTRIBUTE                | DESCRIPTION  |
|--------------------------|--|
| <code>extension</code>   | Required. The file extension to be associated with the application.  |
| <code>description</code> | Required. A description of the file type for use by the shell.   |
| <code>progid</code>      | Required. A name uniquely identifying the file type.   |
| <code>defaultIcon</code> | Required. Specifies the icon to use for files with this extension. The icon file must be specified by using the <a href="#">&lt;file&gt; Element</a> within the <a href="#">&lt;assembly&gt; Element</a> that contains this element. |

## Remarks

This element must include an XML namespace reference to "urn:schemas-microsoft-com:clickonce.v1". If the `<fileAssociation>` element is used, it must come after the `<application>` element in its parent [<assembly> Element](#).

ClickOnce will not overwrite existing file associations. However, a ClickOnce application can override the file extension for the current user only. After that ClickOnce application is uninstalled, ClickOnce deletes the file association for the user, and the per-machine association is active again.

## Example

The following code example illustrates `fileAssociation` elements in an application manifest for a text editor application deployed using ClickOnce. This code example also includes the [<file> Element](#) required by the `defaultIcon` attribute.

```
<file name="text.ico" size="4286">
  <hash>
    <dsig:Transforms>
      <dsig:Transform Algorithm="urn:schemas-microsoft-com:HashTransforms.Identity" />
    </dsig:Transforms>
    <dsig:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
    <dsig:DigestValue>0joAqhmfeBb93ZneZv/oTMP2brY=</dsig:DigestValue>
  </hash>
</file>
<file name="writing.ico" size="9662">
  <hash>
    <dsig:Transforms>
      <dsig:Transform Algorithm="urn:schemas-microsoft-com:HashTransforms.Identity" />
    </dsig:Transforms>
    <dsig:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
    <dsig:DigestValue>2cL2U7cm13nG40v9MQdxYKazIwI=</dsig:DigestValue>
  </hash>
</file>
<fileAssociation xmlns="urn:schemas-microsoft-com:clickonce.v1" extension=".text" description="Text
Document (ClickOnce)" progid="Text.Document" defaultIcon="text.ico" />
<fileAssociation xmlns="urn:schemas-microsoft-com:clickonce.v1" extension=".writing" description="Writings
(ClickOnce)" progid="Writing.Document" defaultIcon="writing.ico" />
```

## See also

- [ClickOnce application manifest](#)



# ClickOnce deployment manifest

3/5/2021 • 2 minutes to read • [Edit Online](#)

A deployment manifest is an XML file that describes a ClickOnce deployment, including the identification of the current ClickOnce application version to deploy.

Deployment manifests have the following elements and attributes.

| ELEMENT  | DESCRIPTION   | ATTRIBUTES   |
|--|---|--|
| <a href="#">&lt;assembly&gt; Element</a>             | Required. Top-level element.  | <code>manifestVersion</code>   |
| <a href="#">&lt;assemblyIdentity&gt; Element</a>     | Required. Identifies the application manifest for the ClickOnce application.  | <code>name</code><br><code>version</code><br><code>publicKeyToken</code><br><code>processorArchitecture</code><br><code>culture</code>                                 |
| <a href="#">&lt;description&gt; Element</a>          | Required. Identifies application information used to create a shell presence and the <b>Add or Remove Programs</b> item in Control Panel. | <code>publisher</code><br><code>product</code><br><code>supportUrl</code>  |
| <a href="#">&lt;deployment&gt; Element</a>           | Optional. Identifies the attributes used for the deployment of updates and exposure to the system.  | <code>install</code><br><code>minimumRequiredVersion</code><br><code>mapFileExtensions</code><br><code>disallowUrlActivation</code><br><code>trustUrlParameters</code> |
| <a href="#">&lt;compatibleFrameworks&gt; Element</a> | Required. Identifies the versions of the .NET Framework where this application can install and run.                                       | <code>SupportUrl</code>  |
| <a href="#">&lt;dependency&gt; Element</a>           | Required. Identifies the version of the application to install for the deployment and the location of the application manifest.           | <code>preRequisite</code><br><code>visible</code><br><code>dependencyType</code><br><code>codebase</code><br><code>size</code>   |

| ELEMENT  | DESCRIPTION   | ATTRIBUTES                               |
|--|---|--|
| <a href="#">&lt;publisherIdentity&gt; Element</a>    | Required for signed manifests. Contains information about the publisher that signed this deployment manifest. | <div>Name</div> <div>issuerKeyHash</div> |
| <a href="#">&lt;Signature&gt; Element</a>            | Optional. Contains the necessary information to digitally sign this deployment manifest.                      | None                                     |
| <a href="#">&lt;customErrorReporting&gt; Element</a> | Optional. Specifies a URI to show when an error occurs.   | Uri                                      |

## Remarks

The deployment manifest file identifies a ClickOnce application deployment, including the current version and other deployment settings. It references the application manifest, which describes the current version of the application and all of the files contained within the deployment.

For more information, see [ClickOnce Security and Deployment](#).

## File location

The deployment manifest file references the correct application manifest for the current version of the application. When you make a new version of an application deployment available, you must update the deployment manifest to refer to the new application manifest.

The deployment manifest file must be strongly named and can also contain certificates for publisher validation.

## File name syntax

The name of a deployment manifest file must end with the *.application* extension.

## Examples

The following code example illustrates a deployment manifest.

```

<?xml version="1.0" encoding="utf-8"?>
<asmv1:assembly xsi:schemaLocation="urn:schemas-microsoft-com:asm.v1 assembly.adaptive.xsd"
  manifestVersion="1.0"
  xmlns:asmv3="urn:schemas-microsoft-com:asm.v3"
  xmlns:dsig=http://www.w3.org/2000/09/xmldsig#
  xmlns:co.v1="urn:schemas-microsoft-com:clickonce.v1"
  xmlns:co.v2="urn:schemas-microsoft-com:clickonce.v2"
  xmlns="urn:schemas-microsoft-com:asm.v2"
  xmlns:asmv1="urn:schemas-microsoft-com:asm.v1"
  xmlns:asmv2="urn:schemas-microsoft-com:asm.v2"
  xmlns:xrml="urn:mpeg:mpeg21:2003:01-REL-R-NS"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <assemblyIdentity
    name="My Application Deployment.app"
    version="1.0.0.0"
    publicKeyToken="43cb1e8e7a352766"
    language="neutral"
    processorArchitecture="x86"
    xmlns="urn:schemas-microsoft-com:asm.v1" />
  <description
    asmv2:publisher="My Company Name"
    asmv2:product="My Application"
    xmlns="urn:schemas-microsoft-com:asm.v1" />
  <deployment install="true">
    <subscription>
      <update>
        <expiration maximumAge="0" unit="days" />
      </update>
    </subscription>
    <deploymentProvider codebase="//myServer/sampleDeployment/MyApplicationDeployment.application" />
  </deployment>
  <compatibleFrameworks xmlns="urn:schemas-microsoft-com:clickonce.v2">
    <framework targetVersion="4.0" profile="Full" supportedRuntime="4.0.20506" />
    <framework targetVersion="4.0" profile="Client" supportedRuntime="4.0.20506" />
  </compatibleFrameworks>
  <dependency>
    <dependentAssembly
      dependencyType="install"
      codebase="1.0.0.0\My Application Deployment.exe.manifest"
      size="6756">
      <assemblyIdentity
        name="My Application Deployment.exe"
        version="1.0.0.0"
        publicKeyToken="43cb1e8e7a352766"
        language="neutral"
        processorArchitecture="x86"
        type="win32" />
      <hash>
        <dsig:Transforms>
          <dsig:Transform Algorithm="urn:schemas-microsoft-com:HashTransforms.Identity" />
        </dsig:Transforms>
        <dsig:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
        <dsig:DigestValue>E506x9FwNauks7UjQywmzgt3FE=</dsig:DigestValue>
      </hash>
    </dependentAssembly>
  </dependency>
  <publisherIdentity name="CN=DOMAIN\MyUsername" issuerKeyHash="18312a18a21b215ecf4c4db20f5a0e0b0dd263c08" />
  <Signature Id="StrongNameSignature" xmlns="http://www.w3.org/2000/09/xmldsig#">
    ...
  </Signature></asmv1:assembly>

```

## See also

- [Publish ClickOnce applications](#)

# <assembly> element (ClickOnce deployment)

3/5/2021 • 2 minutes to read • [Edit Online](#)

The top-level element for the deployment manifest.

## Syntax

```
<assembly  
  manifestVersion  
>
```

## Elements and attributes

The `assembly` element is the root element and is required. Its first contained element must be an `assemblyIdentity` element. The manifest elements must be in the following namespaces: `urn:schemas-microsoft-com:asm.v1`, `urn:schemas-microsoft-com:asm.v2`, and `http://www.w3.org/2000/09/xmldsig#`. Child elements of the assembly must also be in these namespaces, by inheritance or by tagging.

The `assembly` element has the following attribute.

| ATTRIBUTE                    | DESCRIPTION  |
|------------------------------|--|
| <code>manifestVersion</code> | Required. This attribute must be set to <code>1.0</code> . |

## Example

The following code example illustrates an `assembly` element in a deployment manifest for an application deployed using ClickOnce. This code example is part of a larger example provided for the [ClickOnce Deployment Manifest](#) topic.

```
<asmv1:assembly  
  xsi:schemaLocation="urn:schemas-microsoft-com:asm.v1 assembly.adaptive.xsd"  
  manifestVersion="1.0"  
  xmlns:asmv3="urn:schemas-microsoft-com:asm.v3"  
  xmlns:dsig=http://www.w3.org/2000/09/xmldsig#  
  xmlns:co.v1="urn:schemas-microsoft-com:clickonce.v1"  
  xmlns:co.v2="urn:schemas-microsoft-com:clickonce.v2"  
  xmlns="urn:schemas-microsoft-com:asm.v2"  
  xmlns:asmv1="urn:schemas-microsoft-com:asm.v1"  
  xmlns:asmv2="urn:schemas-microsoft-com:asm.v2"  
  xmlns:xrml="urn:mpeg:mpeg21:2003:01-REL-R-NS"  
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

## See also

- [ClickOnce deployment manifest](#)
- [<assembly> element](#)

# <assemblyIdentity> element (ClickOnce deployment)

3/5/2021 • 2 minutes to read • [Edit Online](#)

Identifies the primary assembly of the ClickOnce application.

## Syntax

```
<assemblyIdentity
  name
  version
  publicKeyToken
  processorArchitecture
  type
/>
```

## Elements and attributes

The `assemblyIdentity` element is required. It contains no child elements and has the following attributes.

| ATTRIBUTE                          | DESCRIPTION  |
|------------------------------------|--|
| <code>name</code>                  | <p>Required. Identifies the human-readable name of the deployment for informational purposes.</p> <p>If <code>name</code> contains special characters, such as single or double quotes, the application may fail to activate.</p>  |
| <code>version</code>               | <p>Required. Specifies the version number of the assembly, in the following format: <code>major.minor.build.revision</code>.</p> <p>This value must be incremented in an updated manifest to trigger an application update.</p>  |
| <code>publicKeyToken</code>        | <p>Required. Specifies a 16-character hexadecimal string that represents the last 8 bytes of the SHA-1 hash value of the public key under which the deployment manifest is signed. The public key that is used to sign must be 2048 bits or greater.</p> <p>Although signing an assembly is recommended but optional, this attribute is required. If an assembly is unsigned, you should copy a value from a self-signed assembly or use a "dummy" value of all zeros.</p> |
| <code>processorArchitecture</code> | <p>Required. Specifies the processor. The valid values are <code>msil</code> for all processors, <code>x86</code> for 32-bit Windows, <code>IA64</code> for 64-bit Windows, and <code>Itanium</code> for Intel 64-bit Itanium processors.</p>  |

| ATTRIBUTE         | DESCRIPTION   |
|-------------------|---|
| <code>type</code> | Required. For compatibility with Windows side-by-side installation technology. The only allowed value is <code>win32</code> . |

## Remarks

## Example

The following code example illustrates an `assemblyIdentity` element in a ClickOnce deployment manifest. This code example is part of a larger example provided for the [ClickOnce deployment manifest](#) topic.

```
<!-- Identify the deployment. -->
<assemblyIdentity
  name="My Application Deployment.app"
  version="1.0.0.0"
  publicKeyToken="43cb1e8e7a352766"
  language="neutral"
  processorArchitecture="x86"
  xmlns="urn:schemas-microsoft-com:asm.v1" />
```

## See also

- [ClickOnce deployment manifest](#)
- [<assemblyIdentity> element](#)

# <description> element (ClickOnce deployment)

3/5/2021 • 2 minutes to read • [Edit Online](#)

Identifies application information used to create a shell presence and an **Add or Remove Programs** item in Control Panel.

## Syntax

```
<description
  publisher
  product
  suiteName
  supportUrl
/>
```

## Elements and attributes

The `description` element is required and is in the `urn:schemas-microsoft-com:asm.v1` namespace. It contains no child elements and has the following attributes.

| ATTRIBUTE               | DESCRIPTION  |
|-------------------------|--|
| <code>publisher</code>  | Required. Identifies the company name used for icon placement in the Windows <b>Start</b> menu and the <b>Add or Remove Programs</b> item in Control Panel, when the deployment is configured for install.   |
| <code>product</code>    | Required. Identifies the full product name. Used as the title for the icon installed in the Windows <b>Start</b> menu.   |
| <code>suiteName</code>  | Optional. Identifies a subfolder within the <code>publisher</code> folder in the Windows <b>Start</b> menu.  |
| <code>supportUrl</code> | Optional. Specifies a support URL that is shown in the <b>Add or Remove Programs</b> item in Control Panel. A shortcut to this URL is also created for application support in the Windows <b>Start</b> menu, when the deployment is configured for installation. |

## Remarks

The description element is required in all deployment configurations.

## Example

The following code example illustrates a `description` element in a ClickOnce deployment manifest. This code example is part of a larger example provided for the [ClickOnce Deployment Manifest](#) topic.

```
<description
  asmv2:publisher="My Company Name"
  asmv2:product="My Application"
  xmlns="urn:schemas-microsoft-com:asm.v1" />
```

## See also

- [ClickOnce deployment manifest](#)



# <deployment> element (ClickOnce deployment)

3/5/2021 • 4 minutes to read • [Edit Online](#)

Identifies the attributes used for the deployment of updates and exposure to the system.

## Syntax

```
<deployment
  install
  minimumRequiredVersion
  mapFileExtensions
  disallowUrlActivation
  trustUrlParameters
>
  <subscription>
    <update>
      <beforeApplicationStartup/>
      <expiration
        maximumAge
        unit
      />
    </update>
  </subscription>
  <deploymentProvider
    codebase
  />
</deployment>
```

## Elements and attributes

The `deployment` element is required and is in the `urn:schemas-microsoft-com:asm.v2` namespace. The element has the following attributes.

| ATTRIBUTE                           | DESCRIPTION  |
|-------------------------------------|--|
| <code>install</code>                | Required. Specifies whether this application defines a presence on the Windows <b>Start</b> menu and in the Control Panel <b>Add or Remove Programs</b> application. Valid values are <code>true</code> and <code>false</code> . If <code>false</code> , ClickOnce will always run the latest version of this application from the network, and will not recognize the <code>subscription</code> element.  |
| <code>minimumRequiredVersion</code> | Optional. Specifies the minimum version of this application that can run on the client. If the version number of the application is less than the version number supplied in the deployment manifest, the application will not run. Version numbers must be specified in the format <code>N.N.N.N</code> , where <code>N</code> is an unsigned integer. If the <code>install</code> attribute is <code>false</code> , <code>minimumRequiredVersion</code> must not be set. |

| ATTRIBUTE                          | DESCRIPTION  |
|------------------------------------|--|
| <code>mapFileExtensions</code>     | Optional. Defaults to <code>false</code> . If <code>true</code> , all files in the deployment must have a .deploy extension. ClickOnce will strip this extension off these files as soon as it downloads them from the Web server. If you publish your application by using Visual Studio, it automatically adds this extension to all files. This parameter allows all the files within a ClickOnce deployment to be downloaded from a Web server that blocks transmission of files ending in "unsafe" extensions such as .exe.                                       |
| <code>disallowUrlActivation</code> | Optional. Defaults to <code>false</code> . If <code>true</code> , prevents an installed application from being started by clicking the URL or entering the URL into Internet Explorer. If the <code>install</code> attribute is not present, this attribute is ignored.  |
| <code>trustURLParameters</code>    | <p>Optional. Defaults to <code>false</code>. If <code>true</code>, allows the URL to contain query string parameters that are passed into the application, much like command-line arguments are passed to a command-line application. For more information, see <a href="#">How to: Retrieve Query String Information in an Online ClickOnce Application</a>.</p> <p>If the <code>disallowUrlActivation</code> attribute is <code>true</code>, <code>trustUrlParameters</code> must either be excluded from the manifest, or explicitly set to <code>false</code>.</p> |

The `deployment` element also contains the following child elements.

## subscription

Optional. Contains the `update` element. The `subscription` element has no attributes. If the `subscription` element does not exist, the ClickOnce application will never scan for updates. If the `install` attribute of the `deployment` element is `false`, the `subscription` element is ignored, because a ClickOnce application that is launched from the network always uses the latest version.

## update

Required. This element is a child of the `subscription` element and contains either the `beforeApplicationStartup` or the `expiration` element. `beforeApplicationStartup` and `expiration` cannot both be specified in the same deployment manifest.

The `update` element has no attributes.

## beforeApplicationStartup

Optional. This element is a child of the `update` element and has no attributes. When the `beforeApplicationStartup` element exists, the application will be blocked when ClickOnce checks for updates, if the client is online. If this element does not exist, ClickOnce will first scan for updates based on the values specified for the `expiration` element. `beforeApplicationStartup` and `expiration` cannot both be specified in the same deployment manifest.

## expiration

Optional. This element is a child of the `update` element, and has no children. `beforeApplicationStartup` and

`expiration` cannot both be specified in the same deployment manifest. When the update check occurs and an updated version is detected, the new version caches while the existing version runs. The new version then installs on the next launch of the ClickOnce application.

The `expiration` element supports the following attributes.

| ATTRIBUTE               | DESCRIPTION   |
|-------------------------|---|
| <code>maximumAge</code> | Required. Identifies how old the current update should become before the application performs an update check. The unit of time is determined by the <code>unit</code> attribute. |
| <code>unit</code>       | Required. Identifies the unit of time for <code>maximumAge</code> . Valid units are <code>hours</code> , <code>days</code> , and <code>weeks</code> .                             |

## deploymentProvider

For the .NET Framework 2.0, this element is required if the deployment manifest contains a `subscription` section. For the .NET Framework 3.5 and later, this element is optional, and will default to the server and file path in which the deployment manifest was discovered.

This element is a child of the `deployment` element and has the following attribute.

| ATTRIBUTE             | DESCRIPTION   |
|-----------------------|---|
| <code>codebase</code> | Required. Identifies the location, as a Uniform Resource Identifier (URI), of the deployment manifest that is used to update the ClickOnce application. This element also allows for forwarding update locations for CD-based installations. Must be a valid URI. |

## Remarks

You can configure your ClickOnce application to scan for updates on startup, scan for updates after startup, or never check for updates. To scan for updates on startup, ensure that the `beforeApplicationStartup` element exists under the `update` element. To scan for updates after startup, ensure that the `expiration` element exists under the `update` element, and that update intervals are provided.

To disable checking for updates, remove the `subscription` element. When you specify in the deployment manifest to never scan for updates, you can still manually check for updates by using the [CheckForUpdate](#) method.

For more information on how deploymentProvider relates to updates, see [Choosing a ClickOnce Update Strategy](#).

## Examples

The following code example illustrates a `deployment` element in a ClickOnce deployment manifest. The example uses a `deploymentProvider` element to indicate the preferred update location.

```
<deployment install="true" minimumRequiredVersion="2.0.0.0" mapFileExtension="true"
trustUrlParameters="true">
  <subscription>
    <update>
      <expiration maximumAge="6" unit="hours" />
    </update>
  </subscription>
  <deploymentProvider codebase="http://www.adatum.com/MyApplication.application" />
</deployment>
```

## See also

- [ClickOnce deployment manifest](#)

# <compatibleFrameworks> element (ClickOnce deployment)

3/5/2021 • 2 minutes to read • [Edit Online](#)

Identifies the versions of the .NET Framework where this application can install and run.

## NOTE

*MageUI.exe* does not support the `compatibleFrameworks` element when saving an application manifest that has already been signed with a certificate using *MageUI.exe*. Instead, you must use *Mage.exe*.

## Syntax

```
<compatibleFrameworks
  SupportUrl>
  <framework
    targetVersion
    profile
    supportedRuntime
  />
</ compatibleFrameworks>
```

## Elements and attributes

The `compatibleFrameworks` element is required for deployment manifests that target the ClickOnce runtime provided by .NET Framework 4 or later. The `compatibleFrameworks` element contains one or more `framework` elements that specify the .NET Framework versions on which this application can run. The ClickOnce runtime will run the application on the first available `framework` in this list.

The following table lists the attribute that the `compatibleFrameworks` element supports.

| ATTRIBUTE                              | DESCRIPTION  |
|--|--|
| <code>S</code> <code>supportUrl</code> | Optional. Specifies a URL where the preferred compatible .NET Framework version can be downloaded. |

## framework

Required. The following table lists the attributes that the `framework` element supports.

| ATTRIBUTE                  | DESCRIPTION  |
|----------------------------|--|
| <code>targetVersion</code> | Required. Specifies the version number of the target .NET Framework. |
| <code>profile</code>       | Required. Specifies the profile of the target .NET Framework.        |

| ATTRIBUTE                     | DESCRIPTION  |
|-------------------------------|--|
| <code>supportedRuntime</code> | Required. Specifies the version number of the runtime associated with the target .NET Framework. |

## Remarks

## Example

The following code example shows a `compatibleFrameworks` element in a ClickOnce deployment manifest. This deployment can run on the .NET Framework 4 Client Profile. It can also run on the .NET Framework 4 because it is a superset of the .NET Framework 4 Client Profile.

```
<compatibleFrameworks xmlns="urn:schemas-microsoft-com:clickonce.v2">
  <framework
    targetVersion="4.0"
    profile="Client"
    supportedRuntime="4.0.30319" />
</compatibleFrameworks>
```

## See also

- [ClickOnce deployment manifest](#)

# <dependency> element (ClickOnce deployment)

3/5/2021 • 4 minutes to read • [Edit Online](#)

Identifies the version of the application to install, and the location of the application manifest.

## Syntax

```
<dependency>
<dependentAssembly
  preRequisite
  visible
  dependencyType
  codeBase
  size
>
  <assemblyIdentity
    name
    version
    publicKeyToken
    processorArchitecture
    language
    type
  />
  <hash>
    <dsig:Transforms>
      <dsig:Transform
        Algorithm
      />
    </dsig:Transforms>
    <dsig:DigestMethod />
    <dsig:DigestValue>
    </dsig:DigestValue>
  </hash>

</dependentAssembly>
</dependency>
```

## Elements and attributes

The `dependency` element is required. It has no attributes. A deployment manifest can have multiple `dependency` elements.

The `dependency` element usually expresses dependencies for the main application on assemblies contained within a ClickOnce application. If your Main.exe application consumes an assembly called DotNetAssembly.dll, then that assembly must be listed in a dependency section. Dependency, however, can also express other types of dependencies, such as dependencies on a specific version of the common language runtime, on an assembly in the global assembly cache (GAC), or on a COM object. Because it is a no-touch deployment technology, ClickOnce cannot initiate download and installation of these types of dependencies, but it does prevent the application from running if one or more of the specified dependencies do not exist.

## dependentAssembly

Required. This element contains the `assemblyIdentity` element. The following table shows the attributes the `dependentAssembly` supports.

| ATTRIBUTE                   | DESCRIPTION  |
|-----------------------------|--|
| <code>preRequisite</code>   | Optional. Specifies that this assembly should already exist in the GAC. Valid values are <code>true</code> and <code>false</code> . If <code>true</code> , and the specified assembly does not exist in the GAC, the application fails to run.   |
| <code>visible</code>        | Optional. Identifies the top-level application identity, including its dependencies. Used internally by ClickOnce to manage application storage and activation.  |
| <code>dependencyType</code> | Required. The relationship between this dependency and the application. Valid values are: <ul style="list-style-type: none"> <li>- <code>install</code>. Component represents a separate installation from the current application.</li> <li>- <code>preRequisite</code>. Component is required by the current application.</li> </ul> |
| <code>codebase</code>       | Optional. The full path to the application manifest.   |
| <code>size</code>           | Optional. The size of the application manifest, in bytes.  |

## assemblyIdentity

Required. This element is a child of the `dependentAssembly` element. The content of `assemblyIdentity` must be the same as described in the ClickOnce application manifest. The following table shows the attributes of the `assemblyIdentity` element.

| ATTRIBUTE                          | DESCRIPTION   |
|------------------------------------|---|
| <code>Name</code>                  | Required. Identifies the name of the application.   |
| <code>Version</code>               | Required. Specifies the version number of the application, in the following format: <code>major.minor.build.revision</code>   |
| <code>publicKeyToken</code>        | Required. Specifies a 16-character hexadecimal string that represents the last 8 bytes of the SHA-1 hash of the public key under which the application or assembly is signed. The public key used to sign must be 2048 bits or greater. |
| <code>processorArchitecture</code> | Required. Specifies the microprocessor. The valid values are <code>x86</code> for 32-bit Windows and <code>IA64</code> for 64-bit Windows.  |
| <code>Language</code>              | Optional. Identifies the two part language codes of the assembly. For example, EN-US, which stands for English (U.S.). The default is <code>neutral</code> . This element is in the <code>asmv2</code> namespace.                       |
| <code>type</code>                  | Optional. For backwards compatibility with Windows side-by-side install technology. The only allowed value is <code>win32</code> .  |

## hash

The `hash` element is an optional child of the `file` element. The `hash` element has no attributes.



ClickOnce uses an algorithmic hash of all the files in an application as a security check to ensure that none of the files were changed after deployment. If the `hash` element is not included, this check will not be performed. Therefore, omitting the `hash` element is not recommended.

## dsig:Transforms

The `dsig:Transforms` element is a required child of the `hash` element. The `dsig:Transforms` element has no attributes.

## dsig:Transform

The `dsig:Transform` element is a required child of the `dsig:Transforms` element. The following table shows the attributes of the `dsig:Transform` element.

| ATTRIBUTE              | DESCRIPTION  |
|------------------------|--|
| <code>Algorithm</code> | The algorithm used to calculate the digest for this file. Currently the only value used by ClickOnce is <code>urn:schemas-microsoft-com:HashTransforms.Identity</code> . |

## dsig:DigestMethod

The `dsig:DigestMethod` element is a required child of the `hash` element. The following table shows the attributes of the `dsig:DigestMethod` element.

| ATTRIBUTE              | DESCRIPTION   |
|------------------------|---|
| <code>Algorithm</code> | The algorithm used to calculate the digest for this file. Currently the only value used by ClickOnce is <code>http://www.w3.org/2000/09/xml#sha1</code> . |

## dsig:DigestValue

The `dsig:DigestValue` element is a required child of the `hash` element. The `dsig:DigestValue` element has no attributes. Its text value is the computed hash for the specified file.

## Remarks

Deployment manifests typically have a single `assemblyIdentity` element that identifies the name and version of the application manifest.

## Example 1

The following code example shows a `dependency` element in a ClickOnce deployment manifest.

```

<!-- Identify the assembly dependencies -->
<dependency>
  <dependentAssembly dependencyType="install" allowDelayedBinding="true" codebase="MyApplication.exe"
size="16384">
    <assemblyIdentity name="MyApplication" version="0.0.0.0" cultural="neutral" processorArchitecture="msil"
/>
    <hash>
      <dsig:Transforms>
        <dsig:Transform Algorithm="urn:schemas-microsoft-com:HashTransforms.Identity" />
      </dsig:Transforms>
      <dsig:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
      <dsig:DigestValue>YzXYZJAvj9pgAG3y8jXUjC7AtHg=</dsig:DigestValue>
    </hash>
  </dependentAssembly>
</dependency>

```

## Example 2

The following code example specifies a dependency on an assembly already installed in the GAC.

```

<dependency>
  <dependentAssembly dependencyType="preRequisite" allowDelayedBinding="true">
    <assemblyIdentity name="GACAssembly" version="1.0.0.0" language="neutral" processorArchitecture="msil"
/>
  </dependentAssembly>
</dependency>

```

## Example 3

The following code example specifies a dependency on a specific version of the common language runtime.

```

<dependency>
  <dependentAssembly dependencyType="preRequisite" allowDelayedBinding="true">
    <assemblyIdentity name="Microsoft.Windows.CommonLanguageRuntime" version="2.0.50215.0" />
  </dependentAssembly>
</dependency>

```

## Example 4

The following code example specifies an operating system dependency.

```

<dependency>
  <dependentOS supportUrl="http://www.microsoft.com" description="Microsoft Windows Operating System">
    <osVersionInfo>
      <os majorVersion="4" minorVersion="10" />
    </osVersionInfo>
  </dependentOS>
</dependency>

```

## See also

- [ClickOnce deployment manifest](#)
- [<dependency> element](#)

# <publisherIdentity> element (ClickOnce deployment)

3/5/2021 • 2 minutes to read • [Edit Online](#)

Contains information about the publisher that signed this deployment manifest.

## Syntax

```
<publisherIdentity
  name
  issuerKeyHash
/>
```

## Elements and attributes

The `publisherIdentity` element is required for signed manifests. The following table shows the attributes that the `publisherIdentity` element supports.

| ATTRIBUTE                  | DESCRIPTION  |
|----------------------------|--|
| <code>name</code>          | Required. Describes the identity of the party that published this application. |
| <code>issuerKeyHash</code> | Required. Contains the SHA-1 hash of the public key of the certificate issuer. |

### Parameters

## Property value/return value

## Exceptions

## Remarks

## Requirements

## Subhead

# <Signature> element (ClickOnce deployment)

3/5/2021 • 2 minutes to read • [Edit Online](#)

Contains the necessary information to digitally sign this deployment manifest.

## Syntax

```
<Signature>
  XML signature information
</Signature>
```

## Remarks

Signing a deployment manifest using an envelope signature is optional, but recommended. For more information about signing XML files, see the World Wide Web Consortium Recommendation, "XML-Signature Syntax and Processing," described at <http://www.w3.org/TR/xmlsig-core/>.

If you want to sign your manifest, hashes must be provided for all files. A manifest with files that are not hashed cannot be signed, because users cannot verify the contents of unhashed files.

## Example

The following code example illustrates a `Signature` element in a deployment manifest used in a ClickOnce deployment.

```
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo>
    <CanonicalizationMethod Algorithm=
      "http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
    <SignatureMethod Algorithm=
      "http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
    <Reference URI="">
      <Transforms>
        <Transform Algorithm=
          "http://www.w3.org/2000/09/xmldsig#enveloped-signature" />
      </Transforms>
      <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
      <DigestValue>d2z5AE...</DigestValue>
    </Reference>
  </SignedInfo>
  <SignatureValue>
    4PHj6SaopoLp...
  </SignatureValue>
  <KeyInfo>
    <X509Data>
      <X509Certificate>
        MIIHnTCCBoWgAwIBAgIKJY9+nwAHAAB...
      </X509Certificate>
    </X509Data>
  </KeyInfo>
</Signature>
```

## See also

- [ClickOnce deployment manifest](#)

# <customErrorReporting> element (ClickOnce deployment)

3/5/2021 • 2 minutes to read • [Edit Online](#)

Specifies a URI to show when an error occurs.

## Syntax

```
<customErrorReporting  
    uri  
/>
```

## Remarks

This element is optional. Without it, ClickOnce displays an error dialog box showing the exception stack. If the `customErrorReporting` element is present, ClickOnce will instead display the URI indicated by the `uri` parameter. The target URI will include the outer exception class, the inner exception class, and the inner exception message as parameters.

Use this element to add error reporting functionality to your application. Since the generated URI includes information about the type of error, your Web site can parse that information and display, for example, an appropriate troubleshooting screen.

## Example

The following snippet shows the `customErrorReporting` element, together with the generated URI it might produce.

```
<customErrorReporting uri=http://www.contoso.com/applications/error.asp />
```

Example Generated Error:

http://www.contoso.com/applications/error.asp?

outer=System.Deployment.Application.InvalidDeploymentException&&inner=System.Deployment.Application.InvalidDeploymentException&&msg=The%20application%20manifest%20is%20signed,%20but%20the%20deployment%20manifest%20is%20unsigned.%20Both%20manifests%20must%20be%20either%20signed%20or%20unsigned.

## See also

- [ClickOnce deployment manifest](#)

# ClickOnce unmanaged API reference

3/5/2021 • 2 minutes to read • [Edit Online](#)

ClickOnce unmanaged public APIs from dfshim.dll.

## CleanOnlineAppCache

Cleans or uninstalls all online applications from the ClickOnce application cache.

### Return value

If successful, returns S\_OK; otherwise, returns an HRESULT that represents the failure. If a managed exception occurs, returns 0x80020009 (DISP\_E\_EXCEPTION).

### Remarks

Calling CleanOnlineAppCache will start the ClickOnce service if it is not already running.

## GetDeploymentDataFromManifest

Retrieves deployment information from the manifest and activation URL.

### Parameters

| PARAMETER                                 | DESCRIPTION   | TYPE    |
|---|---|---------|
| <code>pcwzActivationUrl</code>            | A pointer to the <code>ActivationURL</code> .   | LPCWSTR |
| <code>pcwzPathToDeploymentManifest</code> | A pointer to the <code>PathToDeploymentManifest</code> .  | LPCWSTR |
| <code>pwzApplicationIdentity</code>       | A pointer to a buffer to receive a NULL-terminated string that specifies the full application identity returned.  | LPWSTR  |
| <code>pdwIdentityBufferLength</code>      | A pointer to a DWORD that is the length of the <code>pwzApplicationIdentity</code> buffer, in WCHARs. This includes the space for the NULL termination character. | LPDWORD |
| <code>pwzProcessorArchitecture</code>     | A pointer to a buffer to receive a NULL-terminated string that specifies the processor architecture of the application deployment, from the manifest.             | LPWSTR  |
| <code>pdwArchitectureBufferLength</code>  | A pointer to a DWORD that is the length of the <code>pwzProcessorArchitecture</code> buffer, in WCHARs.   | LPDWORD |

| PARAMETER                                   | DESCRIPTION   | TYPE    |
|---|---|---------|
| <code>pwzApplicationManifestCodebase</code> | A pointer to a buffer to receive a NULL-terminated string that specifies the codebase of the application manifest, from the manifest.                                   | LPWSTR  |
| <code>pdwCodebaseBufferLength</code>        | A pointer to a DWORD that is the length of the <code>pwzApplicationManifestCodebase</code> buffer, in WCHARs.   | LPDWORD |
| <code>pwzDeploymentProvider</code>          | A pointer to a buffer to receive a NULL-terminated string that specifies the deployment provider from the manifest, if present. Otherwise, an empty string is returned. | LPWSTR  |
| <code>pdwProviderBufferLength</code>        | A pointer to a DWORD that is the length of the <code>pwzProviderBufferLength</code> .   | LPDWORD |

### Return value

If successful, returns S\_OK; otherwise, returns an HRESULT that represents the failure. Returns HRESULTFROMWIN32(ERROR\_INSUFFICIENT\_BUFFER) if a buffer is too small.

### Remarks

Pointers must not be null. `pcwzActivationUrl` and `pcwzPathToDeploymentManifest` must not be empty.

It is the caller's responsibility to clean up the activation URL. For example, adding escape characters where they are needed or removing the query string.

It is the caller's responsibility to limit the input length. For example, the maximum URL length is 2KB.

## LaunchApplication

Launches or installs an application by using a deployment URL.

### Parameters

| PARAMETER                  | DESCRIPTION   | TYPE    |
|----------------------------|---|---------|
| <code>deploymentUrl</code> | A pointer to a NULL-terminated string that contains the URL of the deployment manifest. | LPCWSTR |
| <code>data</code>          | Reserved for future use. Must be NULL.  | LPVOID  |
| <code>flags</code>         | Reserved for future use. Must be 0.   | DWORD   |

### Return value

If successful, returns S\_OK; otherwise, returns an HRESULT that represents the failure. If a managed exception occurs, returns 0x80020009 (DISP\_E\_EXCEPTION).

## See also



- [CleanOnlineAppCache](#)

# Visual Studio Installer Projects Extension and .NET Core 3.1

3/5/2021 • 2 minutes to read • [Edit Online](#)

Packaging applications as an MSI is often accomplished using the Visual Studio Installer Projects Extension.

You can download the extension here: [Visual Studio Installer Projects](#)

## Update for .NET Core

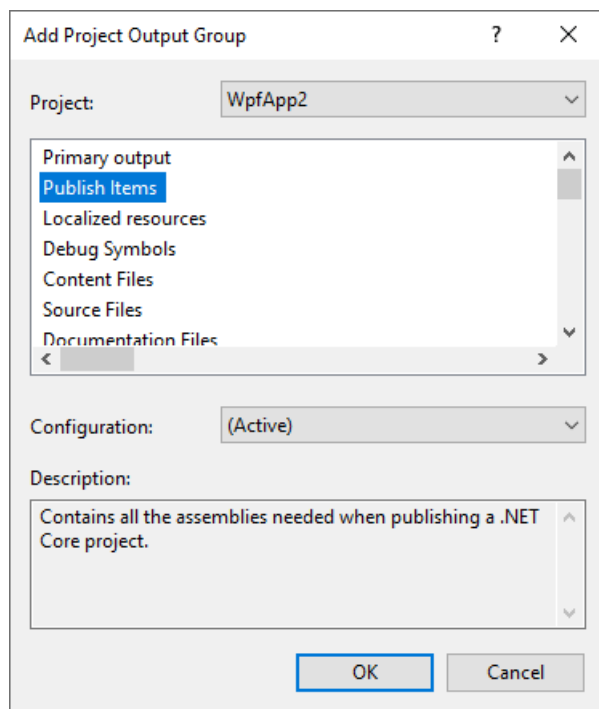
.NET Core has two different models for publishing.

- Framework-dependent deployments
- Self-contained applications include the runtime.

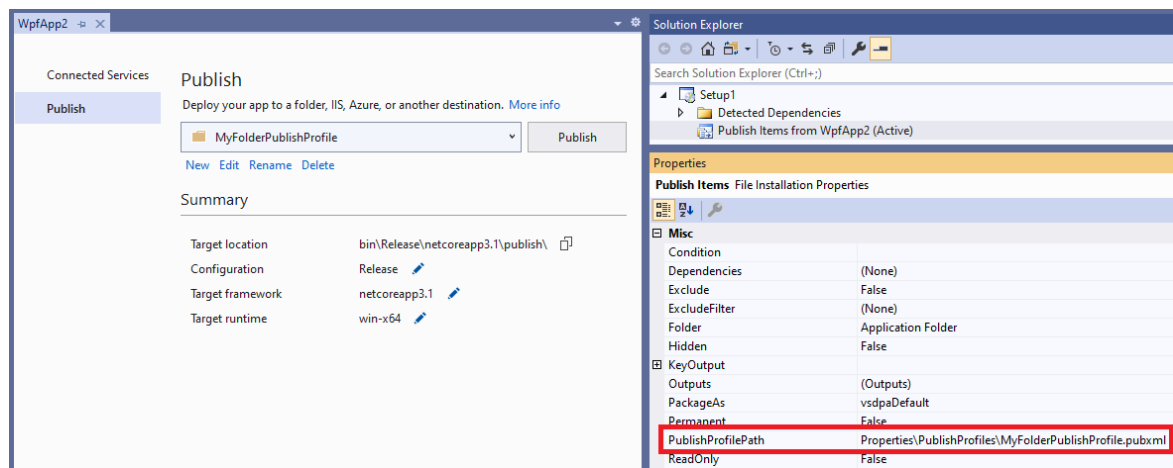
To learn more about these deployment strategies, see [.NET Core application publishing overview](#).

### Workflow changes for .NET Core 3.1

1. Select **Publish Items** instead of **Primary Output** to get the correct output for .NET Core 3.1 projects. To bring up this dialog, select **Add > Project Output...** from the project's context menu.

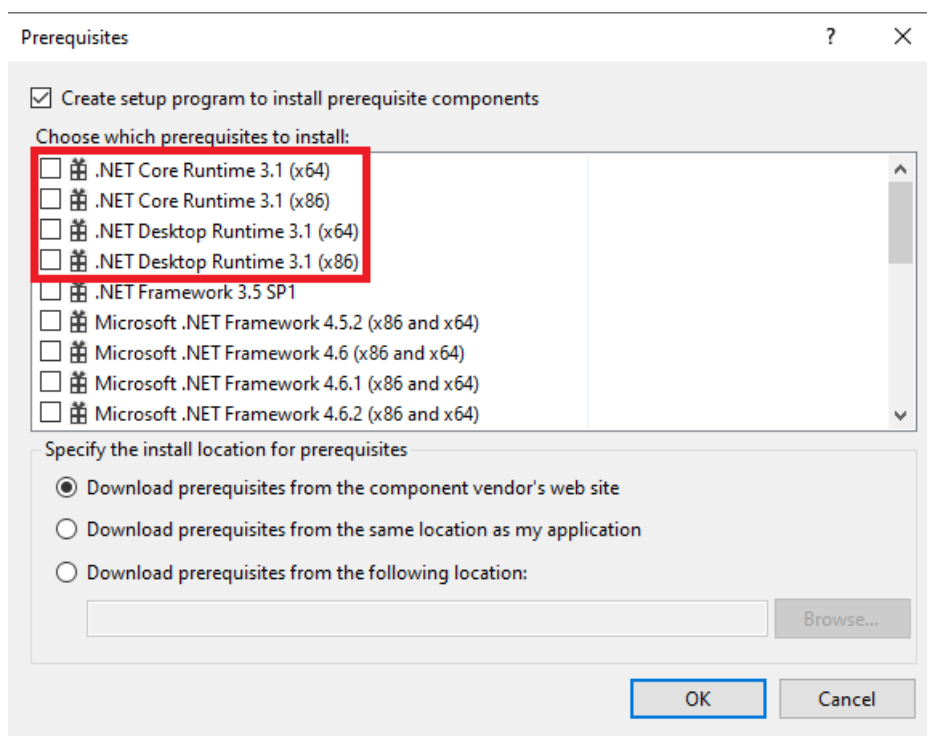


2. To create a self-contained installer, set the **PublishProfilePath** property on the **Publish Items** node in the setup project, using the relative path of a publish profile with the correct properties set.



## Prerequisites for .NET Core 3.1

If you would like your installer to be able to install the necessary runtime for a framework-dependent .NET Core 3.1 app, you can do this using [prerequisites](#). From the properties dialog of your installer project, open the **Prerequisites...** dialog and you'll see the following entries:



The **.NET Core Runtime...** option should be selected for console applications, **.NET Desktop Runtime...** should be selected for WPF/WinForms applications.

### NOTE

These items are present starting with the Visual Studio 2019 Update 7 release.

## See also

- [Prerequisites Dialog Box](#)
- [Application Deployment Prerequisites](#)