Deploying Web Applications in Enterprise Scenarios

Note: This PDF was created from a series of web-based tutorials, the first of which is located at the following URL:

http://www.asp.net/web-forms/tutorials/deployment/deploying-web-applications-in-enterprise-scenarios/deploying-web-applications-in-enterprise-scenarios

Each tutorial includes hyperlinks to others in the series, and those links still go to the web page tutorials, not to pages in this PDF. To go quickly to specific sections of this PDF, use the links in the Table of Contents.

Table of Contents

Introduction

This set of tutorials describes tools and techniques you can use to deploy web applications in various enterprise scenarios. It explains how to make best use of technologies like Visual Studio 2010, the Microsoft Build Engine (MSBuild), Internet Information Services (IIS) 7.5, the IIS Web Deployment Tool (Web Deploy), the Web Farm Framework (WFF), and utilities like VSDBCMD.exe to simplify and manage the deployment process. It includes conceptual overviews and task-oriented guidance that will help you to:

- Review and establish the deployment requirements for an enterprise-scale web application.
- Configure test, staging, and production web server environments to support web deployment.
- Configure Team Foundation Server (TFS) continuous integration (CI) processes to support automated web deployment.
- Deploy enterprise-scale web applications to different server environments with varying requirements and restrictions.
- Deploy changes to web applications that are running in different server environments.

> **Note:** While these tutorials describe the use of TFS as a CI server, the guidance is easily adapted to any CI server. You don't need a detailed knowledge of TFS to understand and leverage the tutorials.

About the Authors

Jason Lee is a principal technologist with Content Master where he has been working with Microsoft products and technologies, especially SharePoint and ASP.NET, for several years. Jason holds a PhD in computing and is currently MCPD and MCTS certified. You can read Jason's technical blog at www.jrjlee.com.

Benjamin Curry is a principal technologist with Content Master who has written whitepapers, SDK documentation, PowerPoint presentations, and instructor-led and online training courses during his career. An original member of the ASP.NET documentation team, he has worked with Microsoft's web technologies for over a decade.

Target Audience

This set of tutorials is for ASP.NET web application developers and solution architects who use Visual Studio 2010 to create enterprise-scale web applications. To get the most value from the content, you should be comfortable using Visual Studio 2010 and have a basic familiarity with TFS, together with an awareness of Microsoft web platform technologies like ASP.NET MVC 3, Windows Communication Foundation (WCF), IIS, SQL Server, and Visual Studio database projects. However, you do not need to be familiar with deployment tools and technologies or need to know how to set up CI systems.

Requirements

To follow the walkthroughs and perform the tasks that these tutorials describe, you'll need to install this software on your development computer:

- Visual Studio 2010 Premium or Ultimate Edition with Service Pack 1
- .NET Framework 4.0
- .NET Framework 3.5 with Service Pack 1
- ASP.NET MVC 3.0
- IIS 7.5 Express
- SQL Server Express 2008 R2

To perform the deployment steps described throughout these walkthroughs, you'll need to have access to sample Web application deployment environments. For best results, these environments should reflect your organization's enterprise deployment pattern. You can then modify the walkthroughs provided in this documentation to reflect the deployment environments and requirements of your own organization.

Series Contents

This introductory section consists of two further topics. These are designed to provide some broader context for the tutorials that follow:

- Enterprise Web Deployment: Scenario Overview. This topic describes the scenario that underpins each of the tutorials in this series. The scenario focuses on the Application Lifecycle Management (ALM) requirements of a fictional company named Fabrikam, Inc. as it develops an enterprise-scale web application.
- Application Lifecycle Management: From Development to Production. This topic provides a high-level, end-to-end overview of a deployment process. It illustrates how Fabrikam,Inc. moves an enterprise-scale ASP.NET web application through test, staging, and production environments as part of a continuous development process.

The series includes four tutorial sets besides these introductory tutorials. Each focuses on different aspects of web deployment:

- Web Deployment in the Enterprise. This tutorial provides a conceptual introduction to MSBuild project files, the Web Publishing Pipeline, Web Deploy, and other related technologies. It explains how you can use these tools together to manage complex deployment processes.
- Configuring Server Environments for Web Deployment. This tutorial describes how to configure Windows servers to support various deployment scenarios, including remote web package deployment using the Web Deployment Agent Service (the "remote agent") or the Web Deploy

Handler and remote database deployment. It provides guidance on choosing the appropriate deployment method for your own environment, and it describes how to use the WFF to replicate deployed web applications across all the web servers in a server farm.

- Configuring Team Foundation Server for Web Deployment. This tutorial describes how to configure TFS to support various deployment scenarios, including automated deployment as part of a CI process and manually triggered deployments of specific builds.
- Advanced Enterprise Web Deployment. This tutorial describes how to accomplish various more advanced deployment tasks, like customizing database deployments for multiple environments, excluding files and folders from deployment, and taking web applications offline during the deployment process.

Where to Start

This set of tutorials uses a sample solution with a realistic level of complexity, together with a fictional enterprise deployment scenario, to provide a reference implementation and to give the tasks and walkthroughs a common context. The next section introduces the scenario and the sample solution. From there you can work through the tutorials and topics that most closely match your needs.

Scenario Overview

This set of tutorials uses a sample solution with a realistic level of complexity, together with a fictional enterprise deployment scenario, to provide a reference implementation and to give the tasks and walkthroughs a common context. This topic describes the tutorial scenario and introduces the sample solution.

Fabrikam, Inc., a fictitious company, is creating a solution that lets remote sales teams store and retrieve contact information from a web interface.

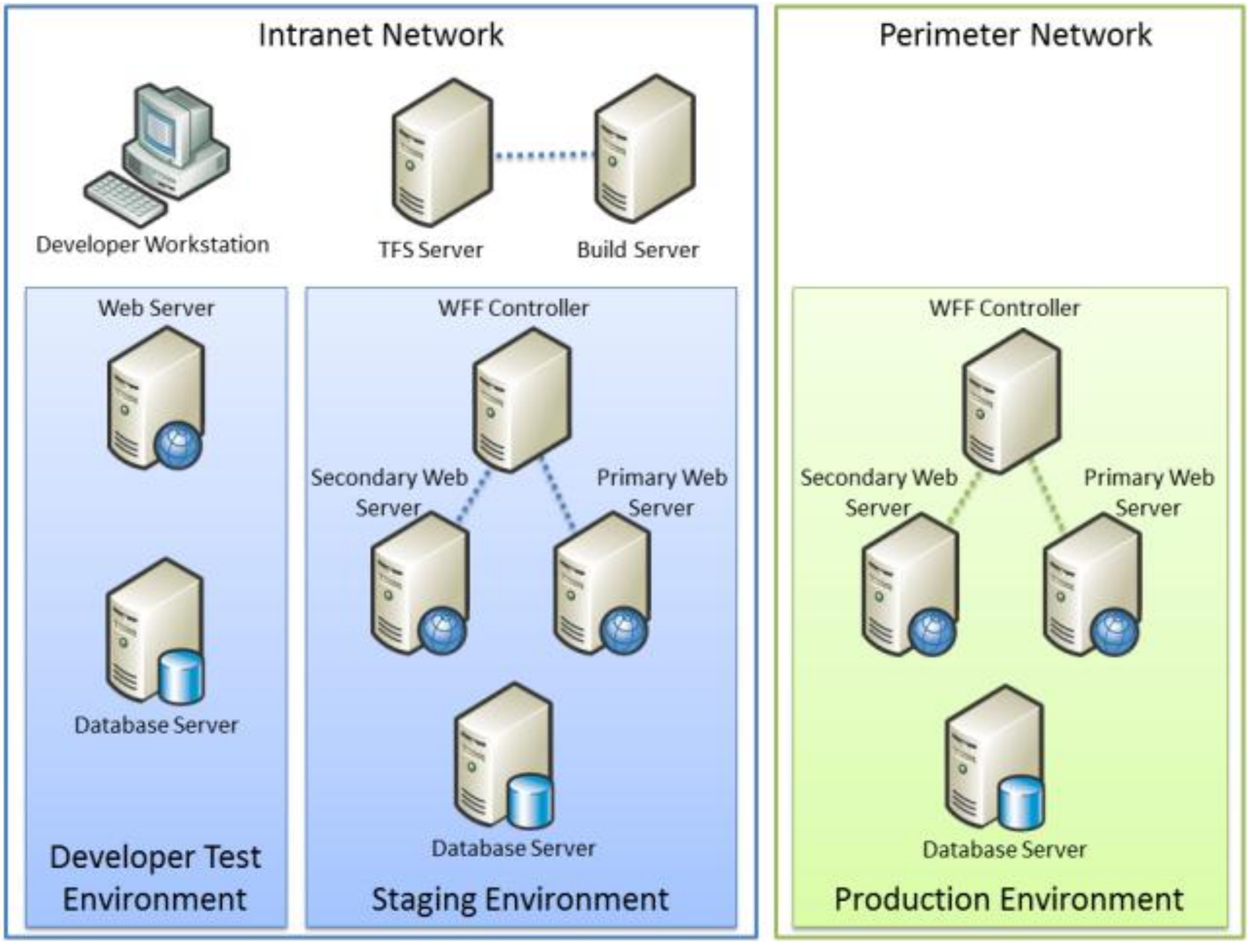
The Application Lifecycle Management (ALM) processes at Fabrikam, Inc. require the solution to be deployed to three server environments at various stages of the software development process:

- A developer test or "sandbox" environment.
- An intranet-based staging environment.
- An Internet-facing production environment.

Each of these environments has different configuration and security requirements, and each poses unique deployment challenges.

The Fabrikam, Inc. Server Infrastructure

This is the high-level development and deployment infrastructure at Fabrikam, Inc.



The developer workstations, the source control infrastructure, the developer test environment, and the staging environment all reside on the intranet network within the Fabrikam.net domain. The production environment resides on a perimeter network (also known as DMZ, demilitarized zone, and screened subnet), which is isolated from the intranet network by a firewall. This is a common deployment scenario: you typically isolate your Internet-facing web servers from your internal server infrastructure through the use of firewalls or gateway servers.

In this example:

- A Team Foundation Server (TFS) 2010 server with a separate build server provides source control and continuous integration (CI) functionality.
- The developer test environment includes an Internet Information Services (IIS) 7.5 web server and a SQL Server 2008 R2 database server.
- The production environment includes multiple IIS 7.5 web servers synchronized by a Web Farm Framework (WFF) controller server, together with a SQL Server 2008 R2 database server. In practice, the database server may use clustering or mirroring to improve scalability and availability.

- The staging environment is designed to replicate the configuration of the production environment as closely as possible.
- The firewall and network isolation policies do not permit direct, automated deployment from the intranet to the perimeter network.

The configuration of each of these environments is described in more detail in the second tutorial, Configuring Server Environments for Web Deployment.

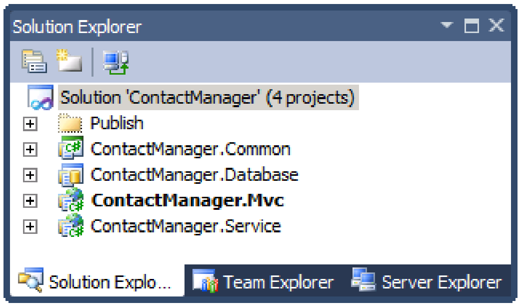
Team Roles for ALM

These users are involved in creating, managing, building, and publishing the Contact Manager solution:

- Matt Hink is a web application developer at Fabrikam, Inc. He is part of the team who developed the Contact Manager solution by using Visual Studio 2010. Matt has full administrator rights on the servers in the developer test environment, which lets him configure the environment to meet his needs. He also has user access to the Visual Studio 2010 TFS instance where he stores the source code for the Contact Manager solution.
- Rob Walters is a server administrator for the Fabrikam, Inc. development team. Rob has administrative access on the TFS server so that he can configure all aspects of TFS and Team Build. Rob also has administrative access to the test and staging web servers and acts as the database administrator (DBA) for the database servers in the test and staging environments. Rob has configured Team Build on the TFS server to carry out these tasks:
    - Build and run unit tests on the application whenever a user checks in a file to TFS. This is called CI.
    - Deploy the Contact Manager application to the test environment automatically once the application passes unit tests. This includes publishing the database to the test servers on initial deployment and any updates to the database after initial deployment.
    - Deploy the Contact Manager application to the staging environment in a single-step process.
    - Create a Web package that a Web server administrator and a DBA can use to publish the application to the production environment.
- Lisa Andrews is a server administrator responsible for deploying applications to the Fabrikam, Inc. production servers. She has read access to the share where the TFS Team Build stores the web deployment package once it builds the Contact Manager application. She also has administrative access to the production web servers so that she can deploy the application to production. Additionally, she acts as the DBA who deploys databases and database updates to the database server in the production environment.

The Contact Manager Solution

The Contact Manager solution is designed to let registered, logged-in users add and edit contact information through a web interface. The Contact Manager solution consists of four individual projects:



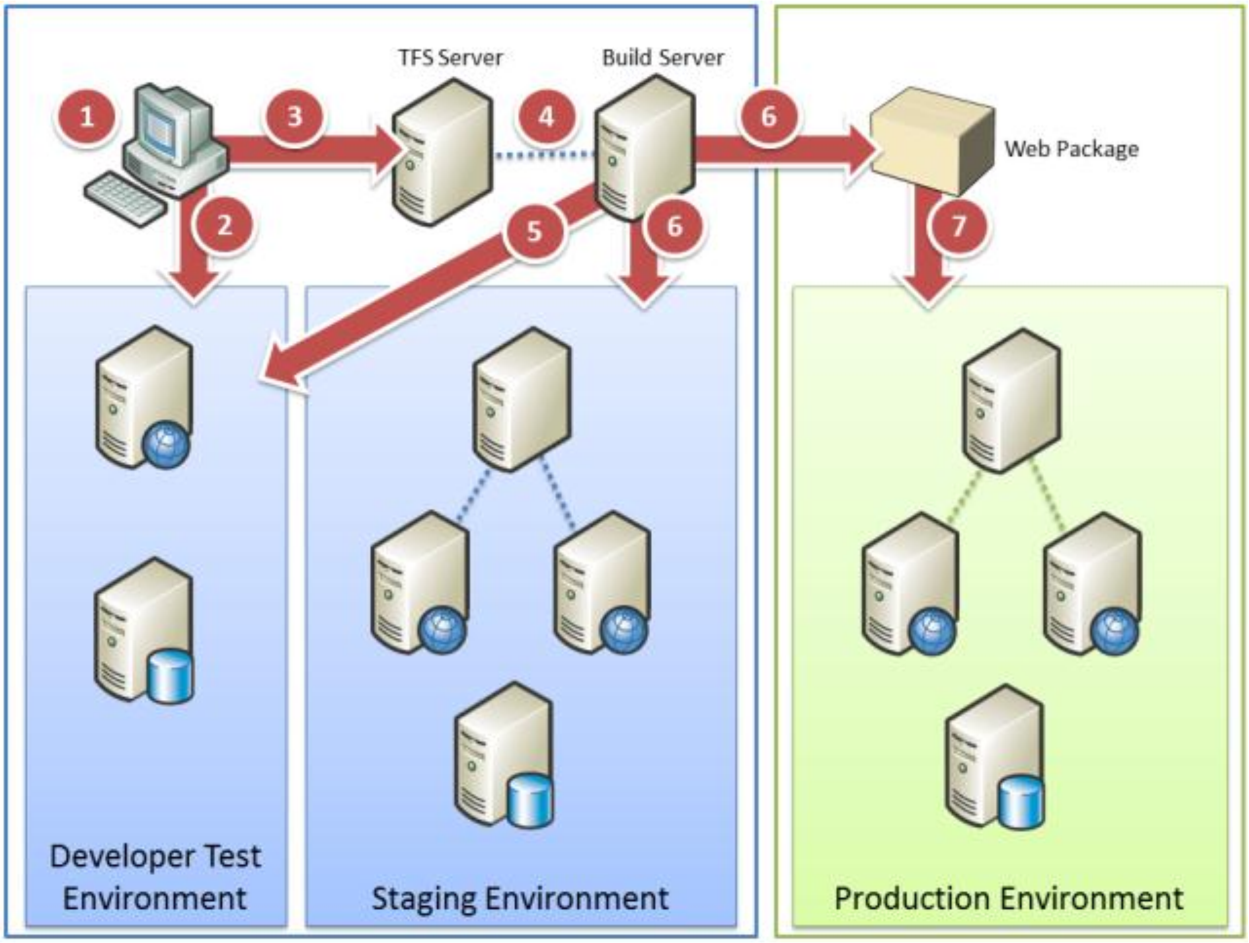
Solution 'ContactManager' (4 projects)
Publish
ContactManager.Common
ContactManager.Database
ContactManager.Mvc
ContactManager.Service

- **ContactManager.Mvc**. This is an ASP.NET MVC3 web application project that represents the entry point for the solution. It offers some basic web application functionality, like providing users with the ability to create and view contact details. The application relies on a Windows Communication Foundation (WCF) service to manage contacts and an ASP.NET application services database to manage authentication and authorization.
- **ContactManager.Database**. This is a Visual Studio 2010 database project. The project defines the schema for a database that stores contact details.
- **ContactManager.Service**. This is a WCF web service project. The WCF exposes an endpoint that allows callers to perform create, retrieve, update, and delete (CRUD) operations on the Contact Manager database. The service relies on the Contact Manager database and the ContactManager.Common.dll assembly.
- **ContactManager.Common**. This is a class library project. The WCF service relies on types defined in this assembly.

A complete review of the solution and its deployment requirements is provided in the first tutorial in this series, Web Deployment in the Enterprise.

Deployment Tasks

There are several distinct tasks involved in deploying applications to different environments in a large organization. These are the key tasks that the tutorials cover:



Here is a list of each step in the deployment process from the perspective of the users described earlier in this document:

1. All members of the team review the Contact Manager solution in Visual Studio 2010 to determine key deployment requirements and issues.
2. Matt Hink may deploy the Contact Manager solution directly from the developer workstation to the developer test environment, to conduct an initial test of the deployment logic.
3. Matt Hink adds the application to source control in TFS.
4. Rob Walters creates various build definitions for the Contact Manager solution in Team Build. One build definition uses CI to deploy the solution to the developer test environment whenever a user checks in new code. Another build definition lets users trigger deployments to the staging environment as required.
5. Every time a user checks in new code, Team Build automatically builds the solution components, runs unit tests, and deploys the solution to the developer test environment if the build was successful and the unit tests pass.

6. When a user triggers a deployment to the staging environment, the solution is packaged and deployed in a single-step process. This process also generates a package for manual deployment to the production environment.
7. Lisa Andrews deploys the application to the production environment by manually importing the web package created in step 6.

Key Deployment Issues

The Contact Manager solution and the Fabrikam, Inc. scenario highlight various common issues and challenges that you may encounter when you deploy complex, enterprise-scale solutions. For example:

- You need to be able to deploy projects to multiple environments, like developer or test environments, staging platforms, and production servers. The solution needs to be deployed with different configuration settings for each environment.
- You need to deploy multiple dependent projects simultaneously as part of a single-step or automated build and deployment process.
- You need to be able to drive deployment from an automated process. For example, you want to use a CI process to deploy web applications to a staging environment when new code is checked in.
- You need to be able to control the deployment process and set deployment variables from outside Visual Studio, as developers are unlikely to have the correct configuration settings or the necessary credentials for every target environment.
- You need to deploy schema-based database projects and preserve existing data on subsequent deployments.
- You need to deploy membership databases on an ad hoc basis without deploying user account data. You may also need to update the schema of deployed membership databases without losing existing user account data.
- You need to exclude certain files or folders when you deploy content to various target environments.

In addition, managing deployment when updates are frequent and incremental throws up some additional challenges. For example:

- You run unit tests every time a developer checks in new code. You only want to deploy the solution if the code passes the unit tests.
- When you deploy a web application to a staging or production environment, you want to redirect users to an *app_offline.htm* file for the duration of the deployment process.
- You want to log deployment activities. The deployment process should send email notifications of successful or failed deployments to designated recipients.

- If an automated deployment fails, the deployment process should retry the current deployment or deploy the previous web package instead.

The next section provides a more detailed look at how Fabrikam, Inc. manages the deployment of the Contact Manager solution to various representative target environments.

Application Lifecycle Management: From Development to Production

This topic illustrates how a fictional company manages the deployment of an ASP.NET web application through test, staging, and production environments as part of a continuous development process. Throughout the topic, links are provided to further information and walkthroughs on how to perform specific tasks.

The topic is designed to provide a high-level overview for a series of tutorials on web deployment in the enterprise. Don't worry if you're not familiar with some of the concepts described here—the tutorials that follow provide detailed information on all of these tasks and techniques.

> **Note:** For the sake of simplicity, this topic doesn't discuss updating databases as part of the deployment process. However, making incremental updates to databases features is a requirement of many enterprise deployment scenarios, and you can find guidance on how to accomplish this later in this tutorial series. For more information, see Deploying Database Projects.

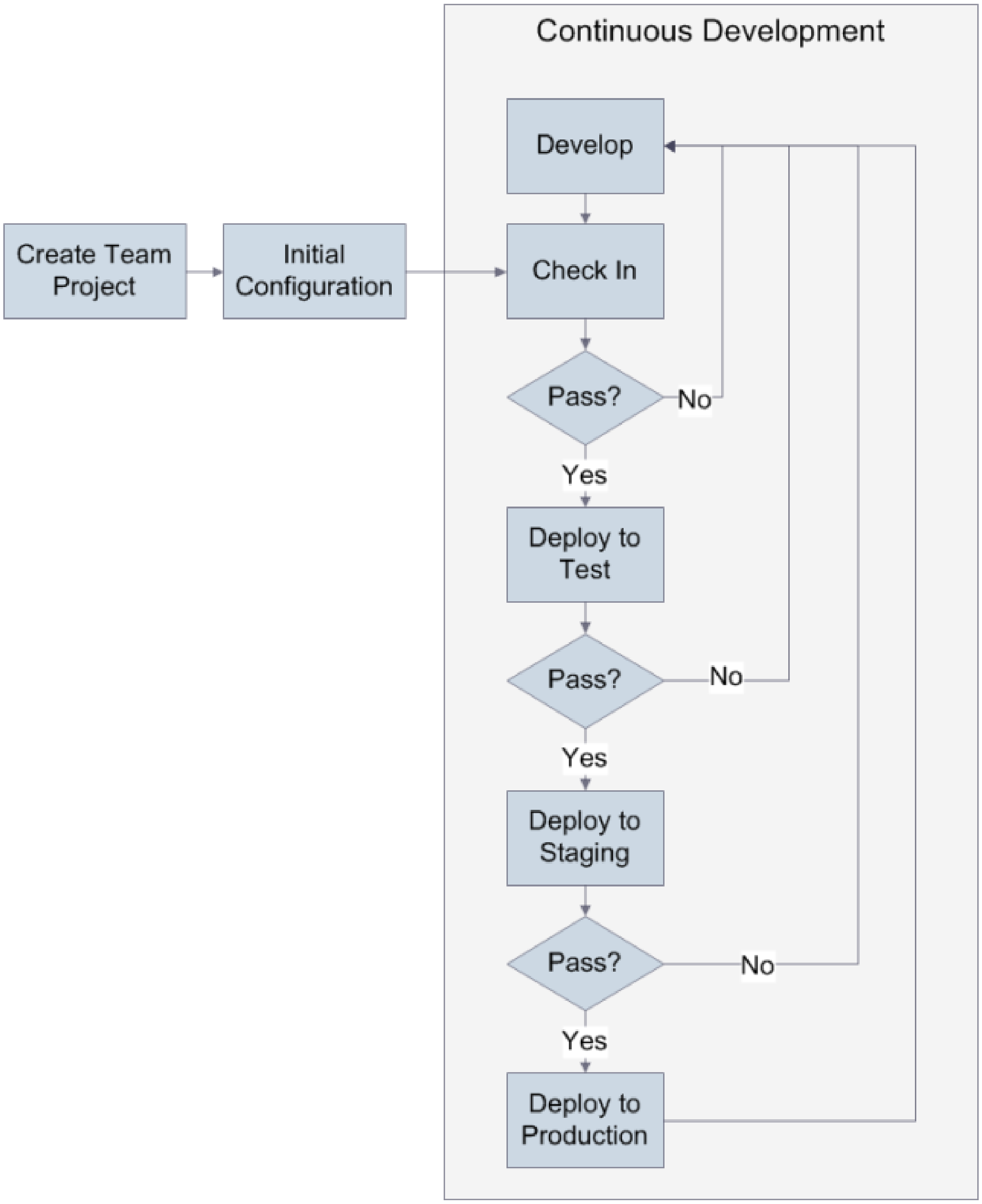
Overview

The deployment process illustrated here is based on the Fabrikam, Inc. deployment scenario described in Enterprise Web Deployment: Scenario Overview. You should read the scenario overview before you study this topic. Essentially, the scenario examines how an organization manages the deployment of a reasonably complex web application, the Contact Manager solution, through various phases in a typical enterprise environment.

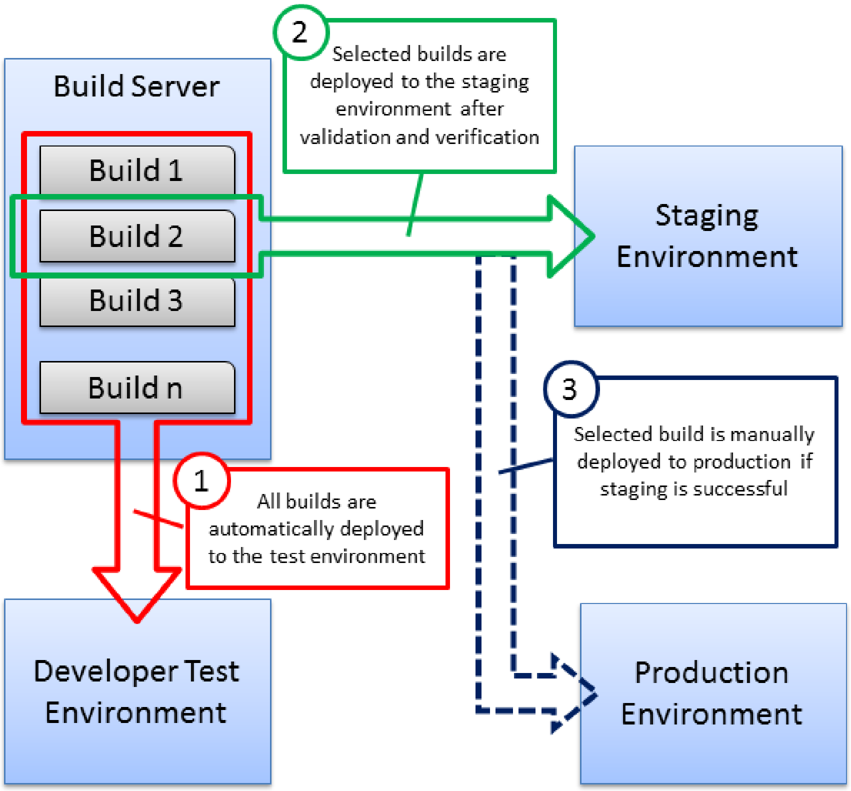
At a high level, the Contact Manager solution goes through these stages as part of the development and deployment process:

1. A developer checks some code into Team Foundation Server (TFS) 2010.
2. TFS builds the code and runs any unit tests associated with the team project.
3. TFS deploys the solution to the test environment.
4. The developer team verifies and validates the solution in the test environment.
5. The staging environment administrator performs a "what if" deployment to the staging environment, to establish whether the deployment will cause any problems.
6. The staging environment administrator performs a live deployment to the staging environment.
7. The solution undergoes user acceptance testing in the staging environment.
8. The web deployment packages are manually imported into the production environment.

These stages form part of a continuous development cycle.



In practice, the process is slightly more complicated than this, as you'll see when we look at each stage in more detail. Fabrikam, Inc. uses a different approach to deployment for each target environment.


Build Server
Build 1
Build 2
Build 3
Build n
2
Selected builds are deployed to the staging environment after validation and verification
Staging Environment
3
Selected build is manually deployed to production if staging is successful
1
All builds are automatically deployed to the test environment
Developer Test Environment
Production Environment

The rest of this topic examines these key stages of this deployment lifecycle:

- **Prerequisites**: How you need to configure your server infrastructure before you put your deployment logic in place.
- **Initial development and deployment**: What you need to do before you deploy your solution for the first time.
- **Deployment to test**: How to package and deploy content to a test environment automatically when a developer checks in new code.
- **Deployment to staging**: How to deploy specific builds to a staging environment and how to perform "what if" deployments to ensure that a deployment won't cause any problems.
- **Deployment to production**: How to import web packages into a production environment when network infrastructure prevents remote deployment.

Prerequisites

The first task in any deployment scenario is to ensure that your server infrastructure meets the requirements of your deployment tools and techniques. In this case, Fabrikam, Inc. has configured its server infrastructure like this:

- TFS is configured to include a team project collection, build controllers, and build agents. See Configuring Team Foundation Server for Automated Web Deployment for more information.

- The test environment is configured to accept remote deployments using the Web Deployment Agent Service (the "remote agent"), as described in Scenario: Configuring a Test Environment for Web Deployment and Configure a Web Server for Web Deploy Publishing (Remote Agent).
- The staging environment is configured to accept remote deployments using the Web Deploy Handler endpoint, as described in Scenario: Configuring a Staging Environment for Web Deployment and Configure a Web Server for Web Deploy Publishing (Web Deploy Handler).
- The production environment is configured to allow an administrator to manually import web deployment packages into Internet Information Services (IIS), as described in Scenario: Configuring a Production Environment for Web Deployment and Configure a Web Server for Web Deploy Publishing (Offline Deployment).

Initial Development and Deployment

Before the Fabrikam, Inc. development team can deploy the Contact Manager solution for the first time, it needs to perform these tasks:

- Create a new team project in TFS.
- Create the Microsoft Build Engine (MSBuild) project files that contain the deployment logic.
- Create the TFS build definitions that trigger the deployment processes.

Create a New Team Project

The TFS administrator, Rob Walters, creates a new team project for the application, as described in Creating a Team Project in TFS. Next, the lead developer, Matt Hink, creates a skeleton solution. He checks his files into the new team project in TFS, as described in Adding Content to Source Control.

Create the Deployment Logic

Matt Hink creates various custom MSBuild project files, using the split project file approach described in Understanding the Project File. Matt creates:

- A project file named *Publish.proj* that runs the deployment process. This file contains MSBuild targets that build the projects in the solution, create web packages, and deploy the packages to a destination server environment.
- Environment-specific project files named *Env-Dev.proj* and *Env-Stage.proj*. These contain settings that are specific to the test environment and the staging environment respectively, like connection strings, service endpoints, and the details of the remote service that will receive the web package. For guidance on choosing the right settings for specific destination environments, see Configure Deployment Properties for a Target Environment.

To run the deployment, a user executes the *Publish.proj* file using MSBuild or Team Build and specifies the location of the relevant environment-specific project file (*Env-Dev.proj* or *Env-Stage.proj*) as a command-line argument. The *Publish.proj* file then imports the environment-specific project file to create a complete set of publishing instructions for each target environment.

> **Note:** The way these custom project files work is independent of the mechanism you use to invoke MSBuild. For example, you can use the MSBuild command line directly, as described in Understanding the Project File. You can run the project files from a command file, as described in Create and Run a Deployment Command File. Alternatively, you can run the project files from a build definition in TFS, as described in Creating a Build Definition that Supports Deployment.
>
> In each case the end result is the same—MSBuild executes the merged project file and deploys your solution to the target environment. This provides you with a great deal of flexibility in how you trigger your publishing process.

Once he has created the custom project files, Matt adds them to a solution folder and checks them into source control.

Create Build Definitions

As a final preparation task, Matt and Rob work together to create three build definitions for the new team project:

- **DeployToTest**. This builds the Contact Manager solution and deploys it to the test environment every time a check-in occurs.
- **DeployToStaging**. This deploys resources from a specified previous build to the staging environment when a developer queues the build.
- **DeployToStaging-WhatIf**. This performs a "what if" deployment to the staging environment when a developer queues the build.

The sections that follow provide more detail on each of these build definitions.

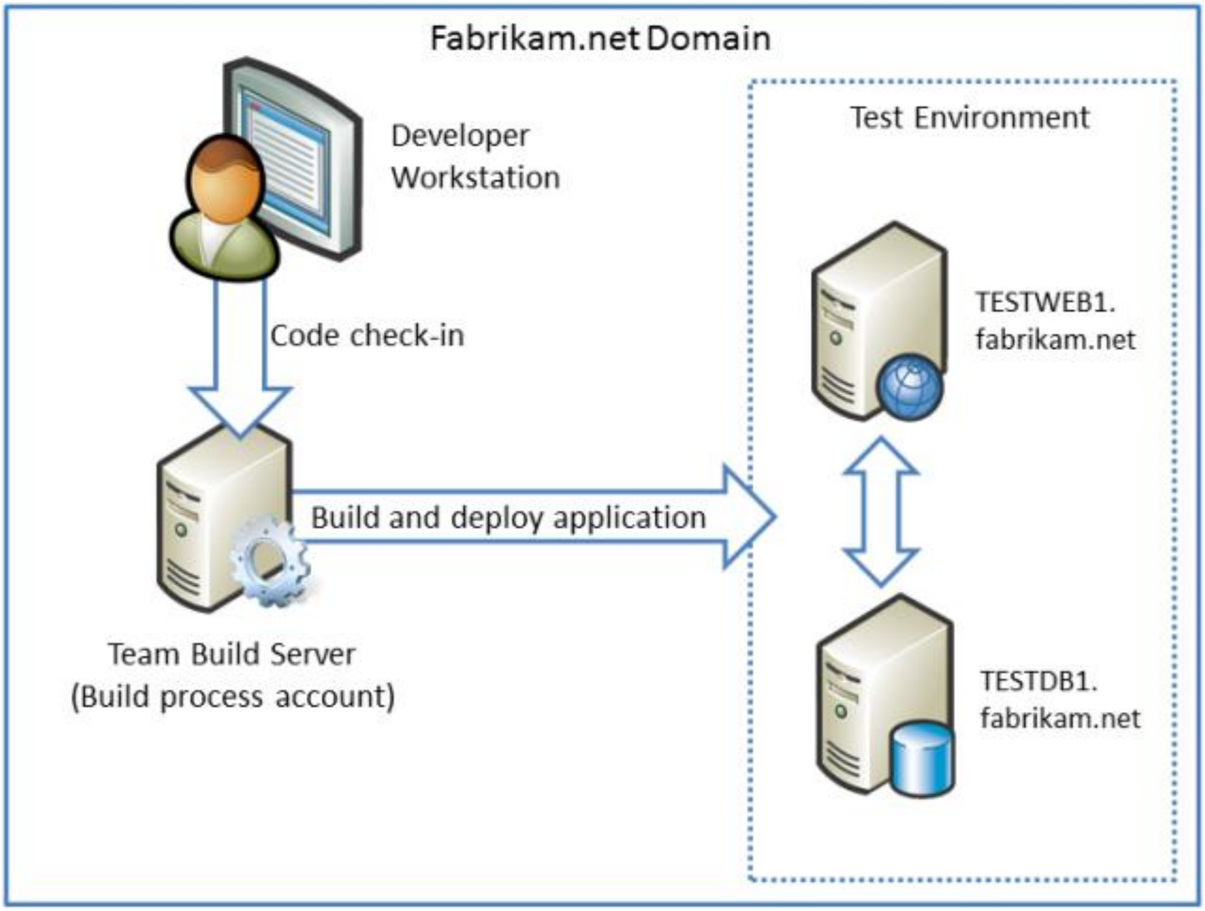
Deployment to Test

The development team at Fabrikam, Inc. maintains test environments to conduct a variety of software testing activities, like verification and validation, usability testing, compatibility testing, and ad hoc or exploratory testing.

The development team has created a build definition in TFS named **DeployToTest**. This build definition uses a continuous integration trigger, which means the build process runs every time a member of the Fabrikam, Inc. development team performs a check-in. When a build is triggered, the build definition will:

- Build the ContactManager.sln solution. This in turn builds every project within the solution.
- Run any unit tests in the solution folder structure (if the solution builds successfully).
- Run the custom project files that control the deployment process (if the solution builds successfully and passes any unit tests).

The end result is that if the solution builds successfully and passes unit tests, the web packages and any other deployment resources are deployed to the test environment.



How Does the Deployment Process Work?

The **DeployToTest** build definition supplies these arguments to MSBuild:

```
/p:DeployOnBuild=true;DeployTarget=package;TargetEnvPropsFile=[path]\Env-Dev.proj
```

The **DeployOnBuild=true** and **DeployTarget=package** properties are used when Team Build builds the projects within the solution. When the project is a web application project, these properties instruct MSBuild to create a web deployment package for the project. The **TargetEnvPropsFile** property tells the *Publish.proj* file where to find the environment-specific project file to import.

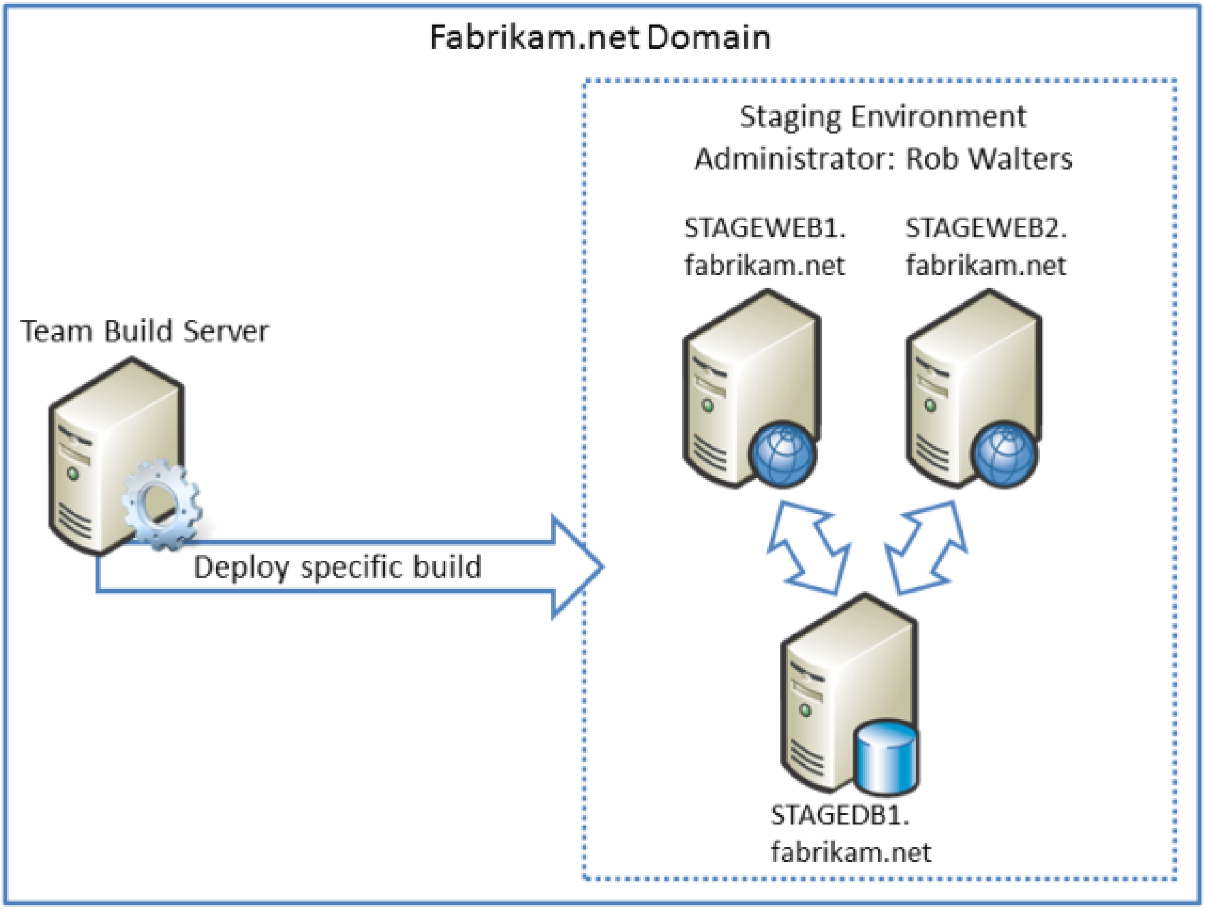
Note: For a detailed walkthrough on how to create a build definition like this, see Creating a Build Definition that Supports Deployment.

The *Publish.proj* file contains targets that build each project in the solution. However, it also includes conditional logic that skips these build targets if you're executing the file in Team Build. This lets you take advantage of the additional build functionality that Team Build offers, like the ability to run unit tests. If the solution build or the unit tests fail, the *Publish.proj* file will not be executed and the application will not be deployed.

The conditional logic is accomplished by evaluating the **BuildingInTeamBuild** property. This is an MSBuild property that is automatically set to **true** when you use Team Build to build your projects.

Deployment to Staging

When a build meets all of the requirements of the developer team in the test environment, the team may want to deploy the same build to a staging environment. Staging environments are typically configured to match the characteristics of the production or "live" environment as closely as possible, for example, in terms of server specifications, operating systems and software, and network configuration. Staging environments are often used for load testing, user acceptance testing, and broader internal reviews. Builds are deployed to the staging environment directly from the build server.



The build definitions used to deploy the solution to the staging environment, **DeployToStaging-WhatIf** and **DeployToStaging**, share these characteristics:

- They don't actually build anything. When Rob deploys the solution to the staging environment, he wants to deploy a specific, existing build that's already been verified and validated in the test environment. The build definitions just need to run the custom project files that control the deployment process.

- When Rob triggers a build, he uses the build parameters to specify which build contains the resources he wants to deploy from the build server.
- The build definitions are not triggered automatically. Rob manually queues a build when he wants to deploy the solution to the staging environment.

This is the high-level process for a deployment to the staging environment:

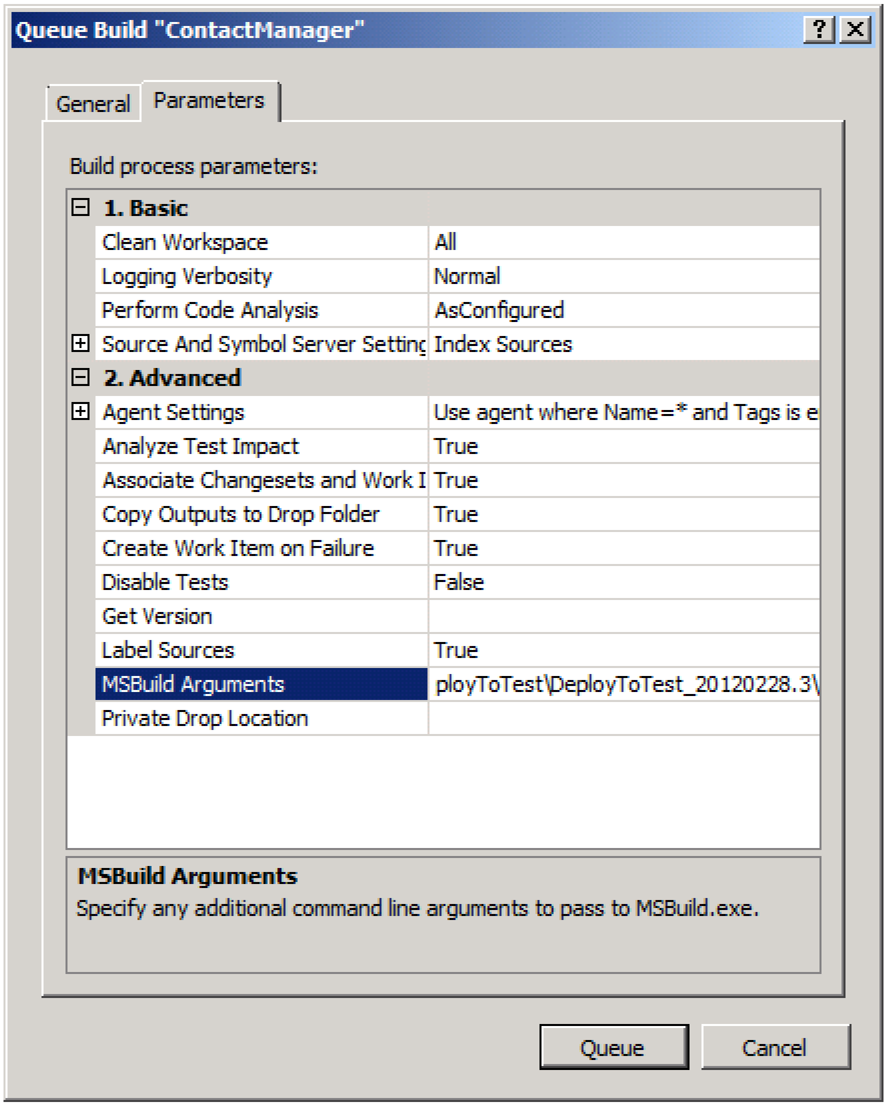
1. The staging environment administrator, Rob Walters, queues a build using the **DeployToStaging-WhatIf** build definition. Rob uses the build definition parameters to specify which build he wants to deploy.
2. The **DeployToStaging-WhatIf** build definition runs the custom project files in "what if" mode. This generates log files as if Rob was performing a live deployment, but it doesn't actually make any changes to the destination environment.
3. Rob reviews the log files to ascertain the effects of the deployment on the staging environment. In particular, Rob wants to check what will be added, what will be updated, and what will be deleted.
4. If Rob is satisfied that the deployment won't make any undesirable changes to existing resources or data, he queues a build using the **DeployToStaging** build definition.
5. The **DeployToStaging** build definition runs the custom project files. These publish the deployment resources to the primary web server in the staging environment.
6. The Web Farm Framework (WFF) controller synchronizes the web servers in the staging environment. This makes the application available on all the web servers in the server farm.

How Does the Deployment Process Work?

The **DeployToStaging** build definition supplies these arguments to MSBuild:

```
/p:TargetEnvPropsFile=[path]\Env-Stage.proj;OutputRoot=[path to build folder]
```

The **TargetEnvPropsFile** property tells the *Publish.proj* file where to find the environment-specific project file to import. The **OutputRoot** property overrides the built-in value and indicates the location of the build folder that contains the resources you want to deploy. When Rob queues the build, he uses the **Parameters** tab to provide an updated value for the **OutputRoot** property.



Build process parameters:

1. Basic	
Clean Workspace	All
Logging Verbosity	Normal
Perform Code Analysis	AsConfigured
Source And Symbol Server Settinç	Index Sources
2. Advanced	
Agent Settings	Use agent where Name=* and Tags is e
Analyze Test Impact	True
Associate Changesets and Work I	True
Copy Outputs to Drop Folder	True
Create Work Item on Failure	True
Disable Tests	False
Get Version	
Label Sources	True
MSBuild Arguments	ployToTest\DeployToTest_20120228.3\
Private Drop Location	

MSBuild Arguments
Specify any additional command line arguments to pass to MSBuild.exe.

Note: For more information on how to create a build definition like this, see Deploy a Specific Build.

The **DeployToStaging-WhatIf** build definition contains the same deployment logic as the **DeployToStaging** build definition. However, it includes the additional argument **WhatIf=true**:

```
/p:TargetEnvPropsFile=[path]\Env-Stage.proj;
   OutputRoot=[path to build folder];
   WhatIf=true
```

Within the *Publish.proj* file, the **WhatIf** property indicates that all deployment resources should be published in "what if" mode. In other words, log files are generated as if the deployment had gone ahead, but nothing is actually changed in the destination environment. This lets you evaluate the impact of a proposed deployment—in particular, what will get added, what will get updated, and what will get deleted—before you actually make any changes.

> **Note:** For more information on how to configure "what if" deployments, see Performing a "What If" Deployment.

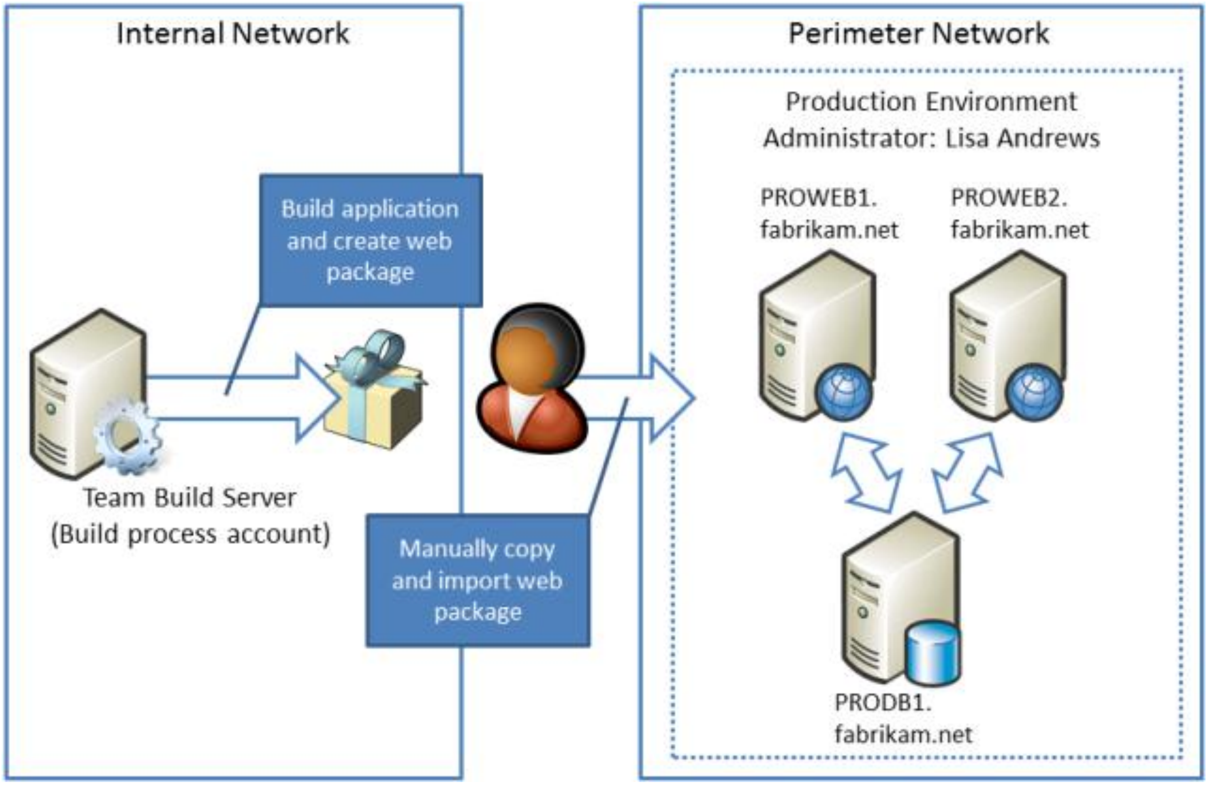
Once you've deployed your application to the primary web server in the staging environment, the WFF will automatically synchronize the application across all the servers in the server farm.

> **Note:** For more information on configuring the WFF to synchronize web servers, see Create a Server Farm with the Web Farm Framework.

Deployment to Production

When a build has been approved in the staging environment, the Fabrikam, Inc. team can publish the application to the production environment. The production environment is where the application goes "live" and reaches its target audience of end users.

The production environment is in an Internet-facing perimeter network. This is isolated from the internal network that contains the build server. The production environment administrator, Lisa Andrews, must manually copy the web deployment packages from the build server and import them into IIS on the primary production web server.



This is the high-level process for a deployment to the production environment:

1. The developer team advises Lisa that a build is ready for deployment to production. The team advises Lisa of the location of the web deployment packages within the drop folder on the build server.

2. Lisa collects the web packages from the build server and copies them to the primary web server in the production environment.
3. Lisa uses IIS Manager to import and publish the web packages on the primary web server.
4. The WFF controller synchronizes the web servers in the production environment. This makes the application available on all the web servers in the server farm.

How Does the Deployment Process Work?

IIS Manager includes an Import Application Package Wizard that makes it easy to publish web packages to an IIS website. For a walkthrough on how to perform this procedure, see Manually Installing Web Packages.

Conclusion

This section provided an illustration of the deployment lifecycle for a typical enterprise-scale web application. There are four chapters (each one is a set of tutorials) in the remainder of this book:

- Web Deployment in the Enterprise. This tutorial provides a conceptual introduction to Microsoft Build Engine (MSBuild) project files, the Web Publishing Pipeline, Web Deploy, and other related technologies. It explains how you can use these tools together to manage complex deployment processes.
- Configuring Server Environments for Web Deployment. This tutorial describes how to configure Windows servers to support various deployment scenarios, including remote web package deployment using the Web Deployment Agent Service (the remote agent) or the Web Deploy Handler and remote database deployment. It provides guidance on choosing the appropriate deployment method for your own environment, and it describes how to use the Web Farm Framework (WFF) to replicate deployed web applications across all the web servers in a server farm.
- Configuring Team Foundation Server for Web Deployment. This tutorial describes how to configure TFS to support various deployment scenarios, including automated deployment as part of a CI process and manually triggered deployments of specific builds.
- Advanced Enterprise Web Deployment. This tutorial describes how to accomplish various more advanced deployment tasks, like customizing database deployments for multiple environments, excluding files and folders from deployment, and taking web applications offline during the deployment process.

Web Deployment in the Enterprise

This tutorial describes how to meet lots of the challenges you'll encounter when you manage the deployment of enterprise-scale web applications to development, test, staging, and production environments. The tutorial includes a reference solution together with a mixture of conceptual and task-oriented content to guide you through various common tasks and procedures.

Enterprise Deployment Challenges

Organizations often encounter these challenges when they look to manage the deployment of complex, enterprise-scale solutions:

- You need to be able to deploy projects to multiple environments, like developer or test environments, staging platforms, and production servers. The solution needs to be deployed with different configuration settings for each environment.
- You need to deploy multiple dependent projects simultaneously as part of a single-step or automated build and deployment process.
- You need to be able to drive deployment from an automated process. For example, you want to use a continuous integration (CI) process to deploy web applications to a test environment when new code is checked in.
- You need to be able to control the deployment process and set deployment variables from outside Visual Studio, as developers are unlikely to have the correct configuration settings or the necessary credentials for every target environment.
- You need to deploy schema-based database projects and preserve existing data on subsequent deployments.
- You need to deploy membership databases on an ad hoc basis without deploying user account data. You may also need to update the schema of deployed membership databases without losing existing user account data.
- You need to exclude certain files or folders when you deploy content to various target environments.

Overview of Approach

This tutorial, together with the other tutorials in this series, uses this high-level approach to meet the challenges described above.

Use custom Microsoft Build Engine (MSBuild) project files to control the overall build and deployment process.

- This lets you build and deploy every project in the solution as part of a single, scriptable operation.

- Environment-specific settings are configured using simple environment-specific project files. In contrast to the Visual Studio–centric approach of using solution configurations and publish profiles to configure deployments for different environments, this approach lets you configure and manage the deployment process from outside Visual Studio. This means that developers don't need advance knowledge of connection strings, service endpoints, server credentials, and other deployment variables for destination environments.
- The custom project files can be invoked by Team Build as part of a Team Foundation Server (TFS) workflow. This lets you configure automated deployment for CI scenarios.

Use the Internet Information Services (IIS) Web Deployment Tool (Web Deploy) to package and deploy web application projects.

- Web Deploy provides a framework that lets you package and deploy your web application content to a destination IIS web server, together with dependencies, configuration settings, security settings, and any other requirements.
- You can control the entire packaging and deployment process from within your custom MSBuild project files. You can also manipulate the configuration settings that accompany your web deployment package, like connection strings, service endpoints, and IIS destination details.
- Web Deploy, together with the Web Publishing Pipeline, offers lots of extensibility points that let you customize your deployments. For example, it's easy to exclude unwanted files and folders from your web deployment packages.

Use the VSDBCMD.exe utility to deploy and update database schemas.

- VSDBCMD allows you to deploy databases from a database schema file (.dbschema), which is generated when you build a Visual Studio database project. In contrast, the database deployment functionality included in Web Deploy is more suited to deploying existing databases from a local SQL Server instance.
- Unlike Visual Studio's functionality for deploying database projects, VSDBCMD lets you deploy differential updates to an existing target database. This allows you to preserve any existing data while you upgrade the database schema.
- You can execute VSDBCMD commands from within your custom MSBuild project files.

Content Map

This tutorial includes topics that fall into four main areas.

These topics introduce the reference solution—the Contact Manager solution—and describe how to download it and configure it on your local machine:

- The Contact Manager Solution

- Setting Up the Contact Manager Solution

These topics introduce MSBuild project files, describe how you can create and use custom project files, and walk through the deployment process for the Contact Manager solution:

- Understanding the Project File
- Understanding the Build Process

These topics describe web application deployment, including how the build and packaging process works, how the build process integrates with the Web Publishing Pipeline, how to modify deployment parameters, and how to deploy web packages to destination environments:

- Building and Packaging Web Application Projects
- Configuring Parameters for Web Package Deployment
- Deploying Web Packages

Deploying Database Projects describes the different techniques you can use to deploy Visual Studio database projects, together with the advantages and disadvantages of each approach. Creating and Running a Deployment Command File describes how to create a simple command file that encapsulates your deployment logic and lets you deploy complex solutions as a single-step process.

Finally, Manually Installing Web Packages concludes the tutorial by showing you to import web packages into IIS.

Key Technologies

The topics in this tutorial primarily use these technologies to manage build and deployment:

- Visual Studio 2010
- MSBuild
- IIS 7.5
- Web Deploy 2.0
- The VSDBCMD.exe database deployment utility

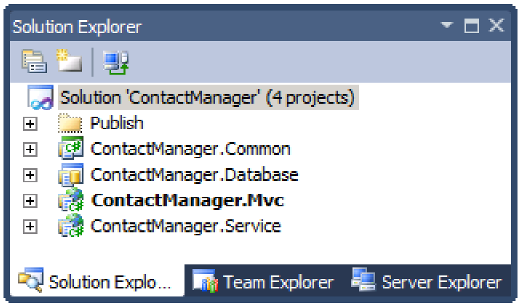
The Contact Manager Solution

This series of tutorials uses a sample solution—the Contact Manager solution—to represent an enterprise-scale application with a realistic level of complexity. This topic introduces the Contact Manager solution, describes the key components of the solution, and identifies the challenges in deploying this kind of application to various destination platforms in an enterprise environment.

As you work through the topics in these tutorials, you can use the Contact Manager solution as a reference implementation that demonstrates how you can meet specific challenges in enterprise deployment scenarios. The next topic, Setting Up the Contact Manager Solution, describes how to download and run the solution on your developer workstation.

Solution Overview

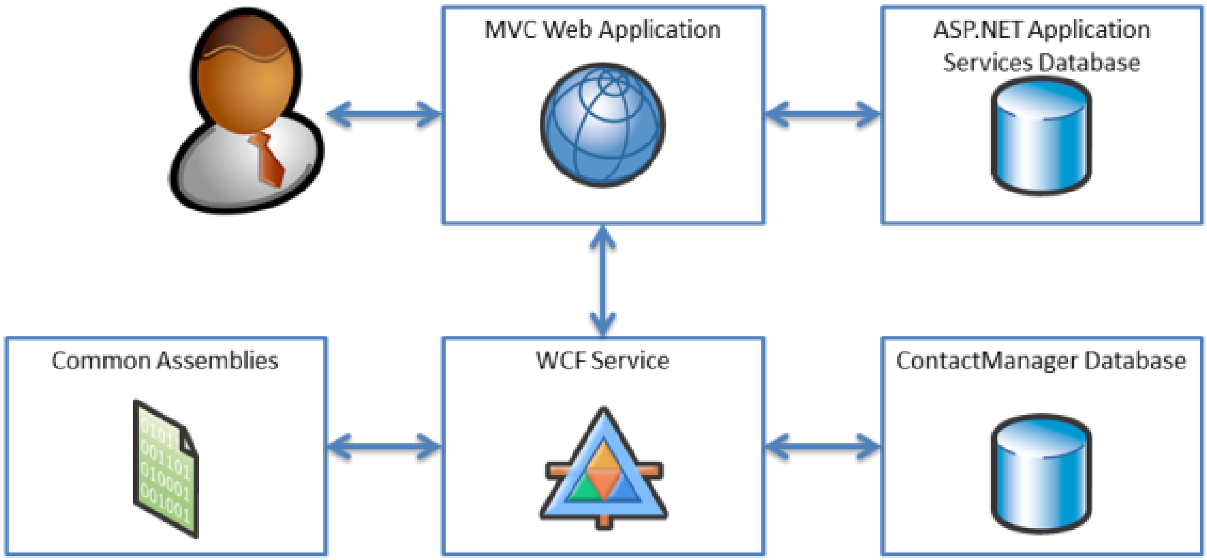
The Contact Manager solution consists of four individual projects:



- **ContactManager.Mvc**. This is an ASP.NET MVC 3 web application project that represents the entry point for the solution. It offers some basic web application functionality, like providing users with the ability to create and view contact details. The application relies on a Windows Communication Foundation (WCF) service to manage contacts and an ASP.NET application services database to manage authentication and authorization.
- **ContactManager.Database**. This is a Visual Studio database project. The project defines the schema for a database that stores contact details.
- **ContactManager.Service**. This is a WCF web service project. The WCF service exposes an endpoint that allows callers to perform create, retrieve, update, and delete (CRUD) operations on the **ContactManager** database. The service relies on the **ContactManager** database and the **ContactManager.Common.dll** assembly.
- **ContactManager.Common**. This is a class library project. The WCF service relies on types defined in this assembly.

The solution also includes a solution folder named Publish. This contains various custom project files and command files that demonstrate how you can control and manipulate the build and deployment process. These are covered in more detail later in this tutorial.

At a conceptual level, the components of the solution fit together like this:



Note: While the ASP.NET MVC 3 web application uses the ASP.NET membership provider, all the pages within the web application allow anonymous access. This is clearly not a realistic configuration. However, the solution is set up in this way to make it easier for you to deploy and test the solution without configuring user accounts and roles.

Deployment Challenges

The Contact Manager solution illustrates several deployment challenges that are common to lots of enterprise deployment scenarios:

- The solution consists of multiple dependent projects. You need to deploy these projects simultaneously.
- Connection strings and service endpoints need to be updated for each environment, and in a lot of cases this information will not be available to the developer.
- When you deploy the **ContactManager** database to staging and production environments, you need to preserve existing data on subsequent deployments.
- When you deploy the ASP.NET application services database, you need to deploy some configuration data but omit any user account data.
- The projects include some files and folders that should not be deployed. You need to exclude these files and folders from the deployment process.
- The solution needs to support automated deployment from a Team Foundation Server (TFS) build server.

Conclusion

This topic provided a high-level overview of the Contact Manager solution and identified some of the inherent deployment challenges that are common to lots of enterprise deployment scenarios. The remaining topics in this tutorial describe some of the techniques you can use to meet these challenges.

The next topic, Setting Up the Contact Manager Solution, describes how to download and run the solution on your developer workstation.

Setting Up the Contact Manager Solution

This topic describes how to download and configure the Contact Manager solution to run locally on a developer workstation.

This topic forms part of a tutorial on web deployment in enterprise scenarios.

System Requirements

To run the Contact Manager solution locally and to perform the other tasks described in this tutorial, you'll need to install this software on your developer workstation:

- Visual Studio 2010 Service Pack 1, Premium or Ultimate Edition
- Internet Information Services (IIS) 7.5 Express
- SQL Server Express 2008 R2
- IIS Web Deployment Tool (Web Deploy) 2.1 or later
- ASP.NET 4.0
- ASP.NET MVC 3
- .NET Framework 4
- .NET Framework 3.5 SP1

With the exception of Visual Studio 2010, you can download and install the latest versions of all of these products and components through the Web Platform Installer.

Download and Extract the Solution

You can download the Contact Manager sample application from the MSDN Code Gallery at the following URL:

http://code.msdn.microsoft.com/Deploying-Web-Applications-9d9093c0

Configure and Run the Solution

To configure and run the Contact Manager solution on your local machine, you'll need to perform these high-level steps:

1. If you don't have one already, create a local ASP.NET application services database with the membership and role management features enabled.
2. Edit connection strings in the *web.config* files to point to your local SQL Server Express instance.
3. Run the solution from Visual Studio 2010.

The remainder of this section provides more guidance on how to complete each of these tasks.

To create the application services database

1. Open a Visual Studio 2010 command prompt. To do this, on the **Start** menu, point to **All Programs**, click **Microsoft Visual Studio 2010**, click **Visual Studio Tools**, and then click **Visual Studio Command Prompt (2010)**.
2. At the command prompt, type this command, and then press Enter:

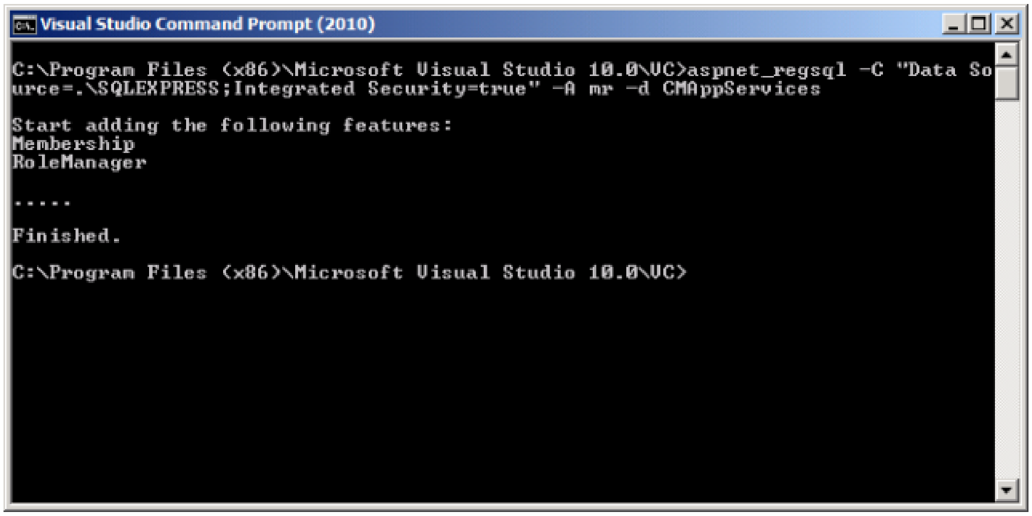
```
aspnet_regsql –C "Data Source=.\SQLEXPRESS;Integrated Security=true" –A mr –d
CMAppServices
```

   a. Use the **–C** switch to specify the connection string for your database server.

   b. Use the **–A** switch to specify the application services features you want to add to the database. In this case, **m** indicates that you want to add support for the membership provider and **r** indicates that you want to add support for the role manager.

   c. Use the **–d** switch to specify a name for your application services database. If you omit this switch, the utility will create a database with the default name of **aspnetdb**.

3. When the database has been created successfully, the command prompt will show a confirmation.



```
C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC>aspnet_regsql -C "Data So
urce=.\SQLEXPRESS;Integrated Security=true" -A mr -d CMAppServices

Start adding the following features:
Membership
RoleManager

.....

Finished.

C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC>
```

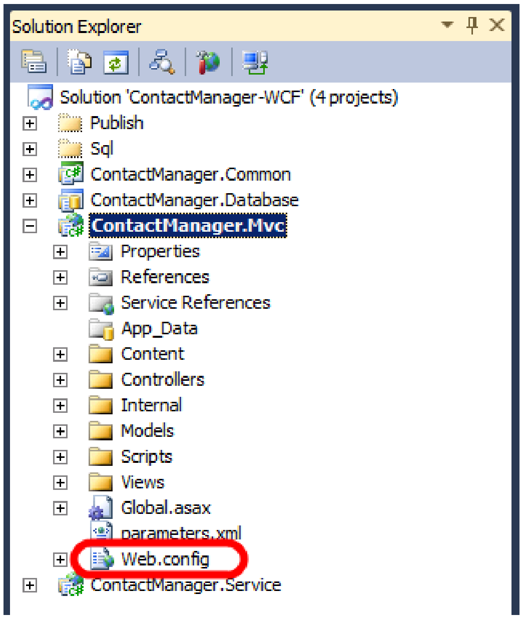
Note: For more information on the aspnet_regsql utility, see ASP.NET SQL Server Registration Tool (Aspnet_regsql.exe).

The next step is to make sure that the connection strings in the Contact Manager solution point to your local instance of SQL Server Express.

To update the connection strings

1. Open the Contact Manager solution in Visual Studio 2010.
2. In the **Solution Explorer** window, expand the **ContactManager.Mvc** project, and then double-click the **Web.config** node.

> **Note:** The ContactManager.Mvc project includes two *web.config* files. You need to edit the project-level file.

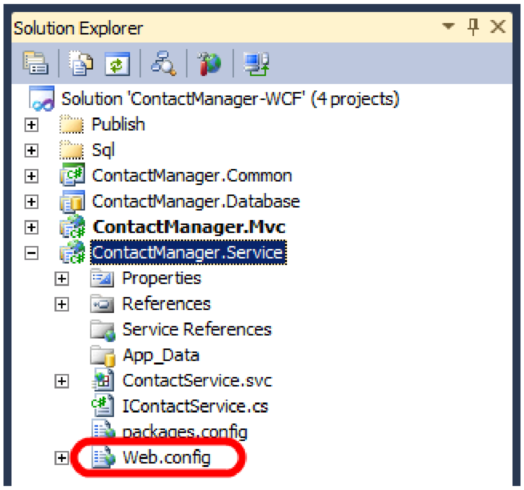


3. In the **connectionStrings** element, verify that the connection string named **ApplicationServices** points to your local ASP.NET application services database.

XML

```xml
<connectionStrings>
  <add name="ApplicationServices"
       connectionString="Data Source=.\SQLEXPRESS;
                         Integrated Security=true;
                         Initial Catalog=CMAppServices"
       providerName="System.Data.SqlClient" />
</connectionStrings>
```

4. In the **Solution Explorer** window, expand the **ContactManager.Service** project, and then double-click the **Web.config** node.



Solution Explorer

Solution 'ContactManager-WCF' (4 projects)

- Publish
- Sql
- ContactManager.Common
- ContactManager.Database
- **ContactManager.Mvc**
- ContactManager.Service
  - Properties
  - References
  - Service References
  - App_Data
  - ContactService.svc
  - IContactService.cs
  - packages.config
  - Web.config

5. In the **connectionStrings** element, in the connection string named **ContactManagerContext**, verify that the **Data Source** property is set to your local instance of SQL Server Express. You don't need to change anything else in the connection string.

XML

```xml
<connectionStrings>
  <add name="ContactManagerContext"
       connectionString="Data Source=.\SQLExpress;
                         Initial Catalog=ContactManager;
                         Integrated Security=true;
                         multipleactiveresultsets=true"
       providerName="System.Data.SqlClient" />
</connectionStrings>
```

6. Save all open files.

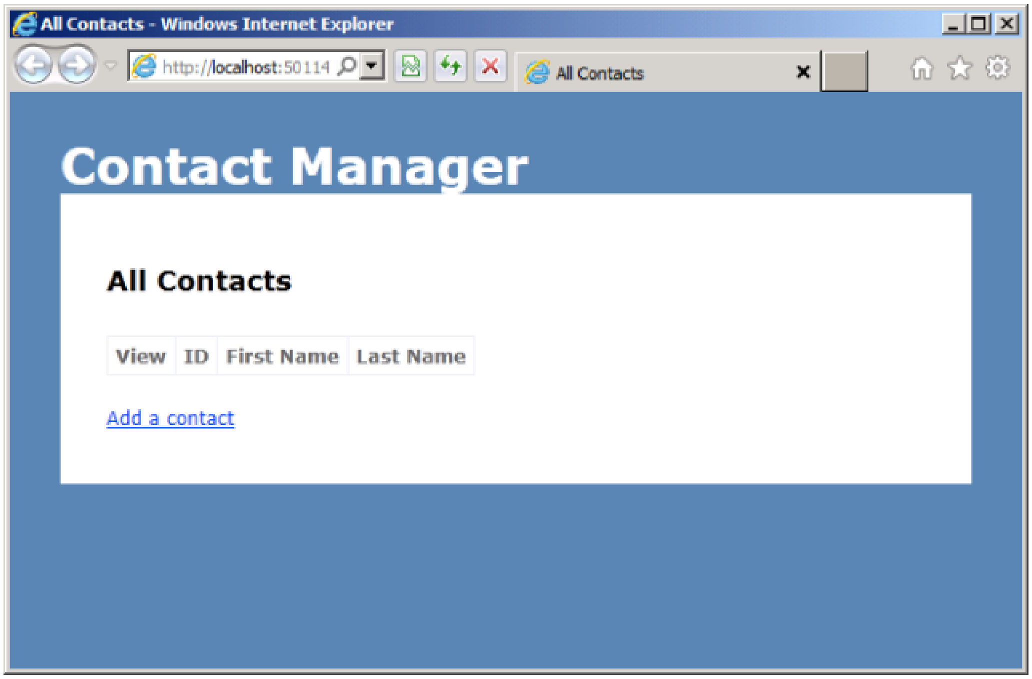
You should now be ready to run the Contact Manager solution on your local machine.

Note: If you follow these steps without first creating an application services database, ASP.NET will create the database the first time you attempt to create a user. However, manually creating the database gives you a lot more control over the application services feature set you want to support.

To run the Contact Manager solution

1. In Visual Studio 2010, press F5.

   Internet Explorer starts up and requests the URL of the Contact Manager ASP.NET MVC 3 application. By default, the application displays the **All Contacts** page.



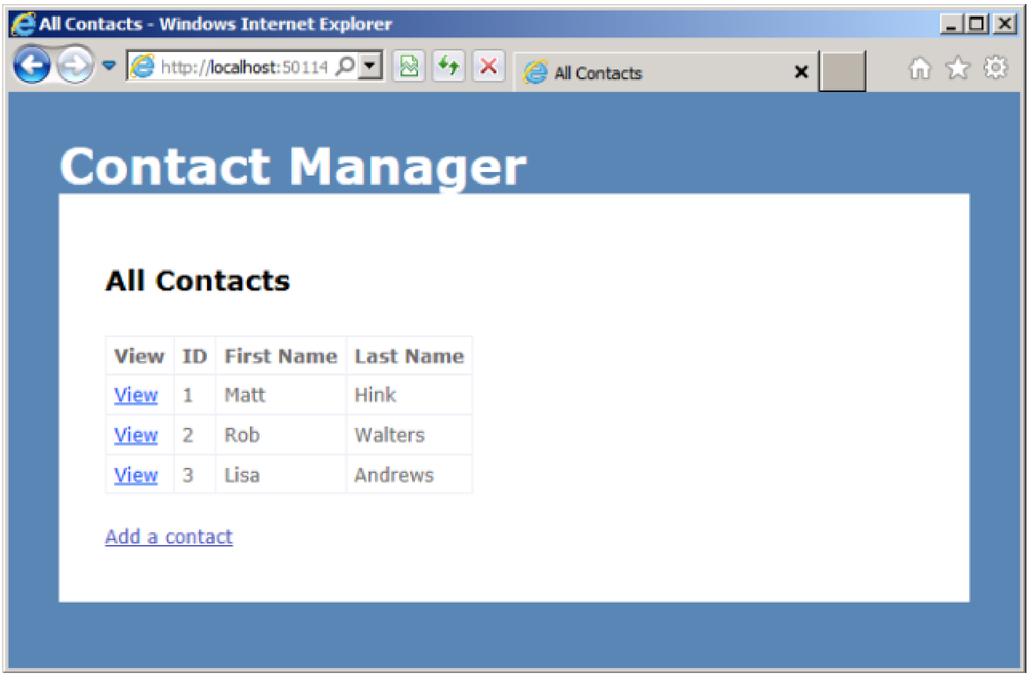
Contact Manager

All Contacts

View	ID	First Name	Last Name

Add a contact

2. Add a few contacts, and then verify that the application works as expected.



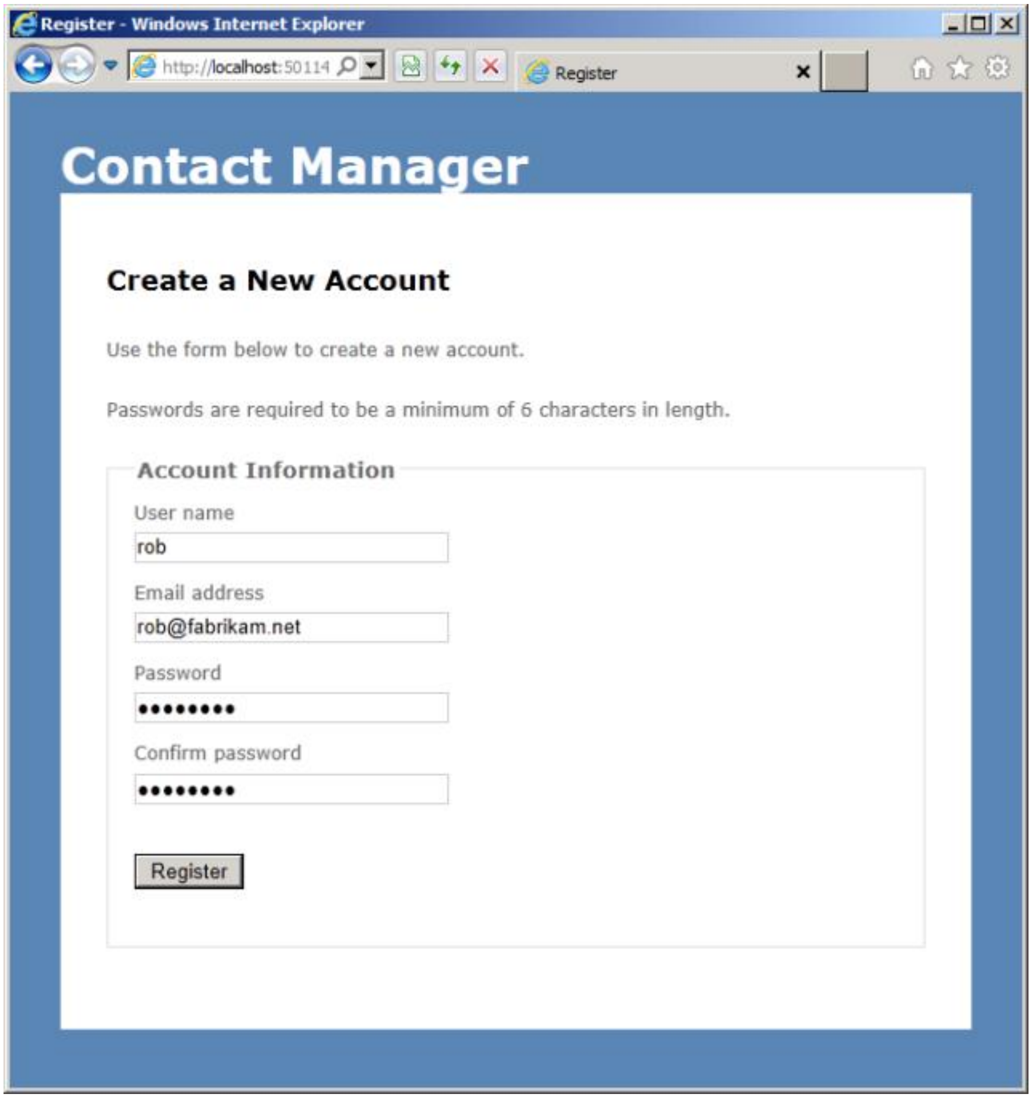
Contact Manager

All Contacts

View	ID	First Name	Last Name
View	1	Matt	Hink
View	2	Rob	Walters
View	3	Lisa	Andrews

Add a contact

3. Browse to **http://localhost:50114/Account/Register** (adjust the URL if you're hosting the application on a different port). Add a user name, email address, and password, and verify that you're able to register an account successfully.



Contact Manager

Create a New Account

Use the form below to create a new account.

Passwords are required to be a minimum of 6 characters in length.

Account Information

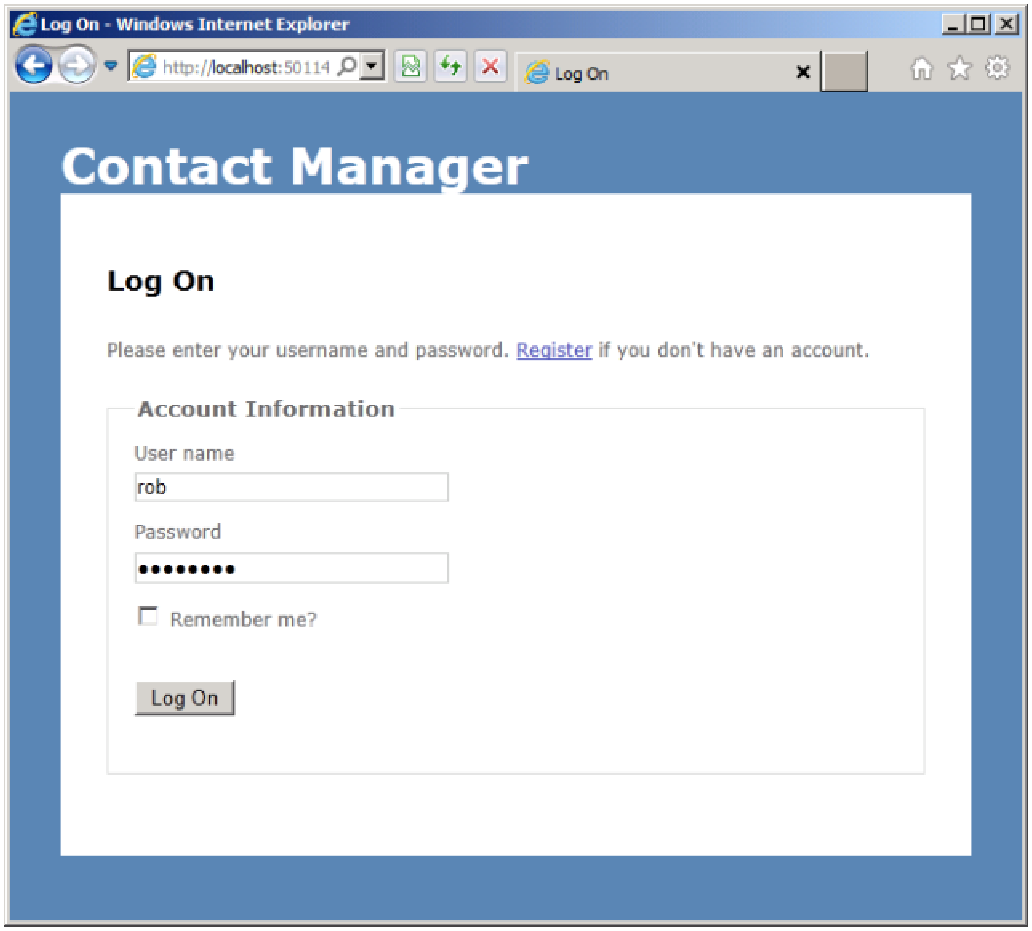
User name: rob

Email address: rob@fabrikam.net

Password: •••••••••

Confirm password: •••••••••

4. Browse to **http://localhost:50114/Account/LogOn** (adjust the URL if you're hosting the application on a different port). Verify that you're able to log on using the account you just created.



Contact Manager

Log On

Please enter your username and password. Register if you don't have an account.

Account Information

User name: rob

Password: ••••••••

□ Remember me?

5. Close Internet Explorer to stop debugging.

Conclusion

At this point, the Contact Manager solution should be fully configured to run on your local machine. You can use the solution as a reference when you work through the other topics in this tutorial.

The next topic, Understanding the Project File, explains how you can use the custom Microsoft Build Engine (MSBuild) project files within the Contact Manager solution to control the deployment process.

Understanding the Project File

Microsoft Build Engine (MSBuild) project files lie at the heart of the build and deployment process. This topic starts with a conceptual overview of MSBuild and the project file. It describes the key components you'll come across when you work with project files, and it works through an example of how you can use project files to deploy real-world applications.

What you'll learn:

- How MSBuild uses MSBuild project files to build projects.

- How MSBuild integrates with deployment technologies, like the Internet Information Services (IIS) Web Deployment Tool (Web Deploy).
- How to understand the key components of a project file.
- How you can use project files to build and deploy complex applications.

This topic forms part of a tutorial on web deployment in the enterprise.

MSBuild and the Project File

When you create and build solutions in Visual Studio, Visual Studio uses MSBuild to build each project in your solution. Every Visual Studio project includes an MSBuild project file, with a file extension that reflects the type of project—for example, a C# project (.csproj), a Visual Basic.NET project (.vbproj), or a database project (.dbproj). In order to build a project, MSBuild must process the project file associated with the project. The project file is an XML document that contains all the information and instructions that MSBuild needs in order to build your project, like the content to include, the platform requirements, versioning information, web server or database server settings, and the tasks that must be performed.

MSBuild project files are based on the MSBuild XML schema, and as a result the build process is entirely open and transparent. In addition, you don't need to install Visual Studio in order to use the MSBuild engine—the MSBuild.exe executable is part of the .NET Framework, and you can run it from a command prompt. As a developer, you can craft your own MSBuild project files, using the MSBuild XML schema, to impose sophisticated and fine-grained control over how your projects are built and deployed. These custom project files work in exactly the same way as the project files that Visual Studio generates automatically.

Note: You can also use MSBuild project files with the Team Build service in Team Foundation Server (TFS). For example, you can use project files in continuous integration (CI) scenarios to automate deployment to a test environment when new code is checked in. For more information, see Configuring Team Foundation Server for Automated Web Deployment.

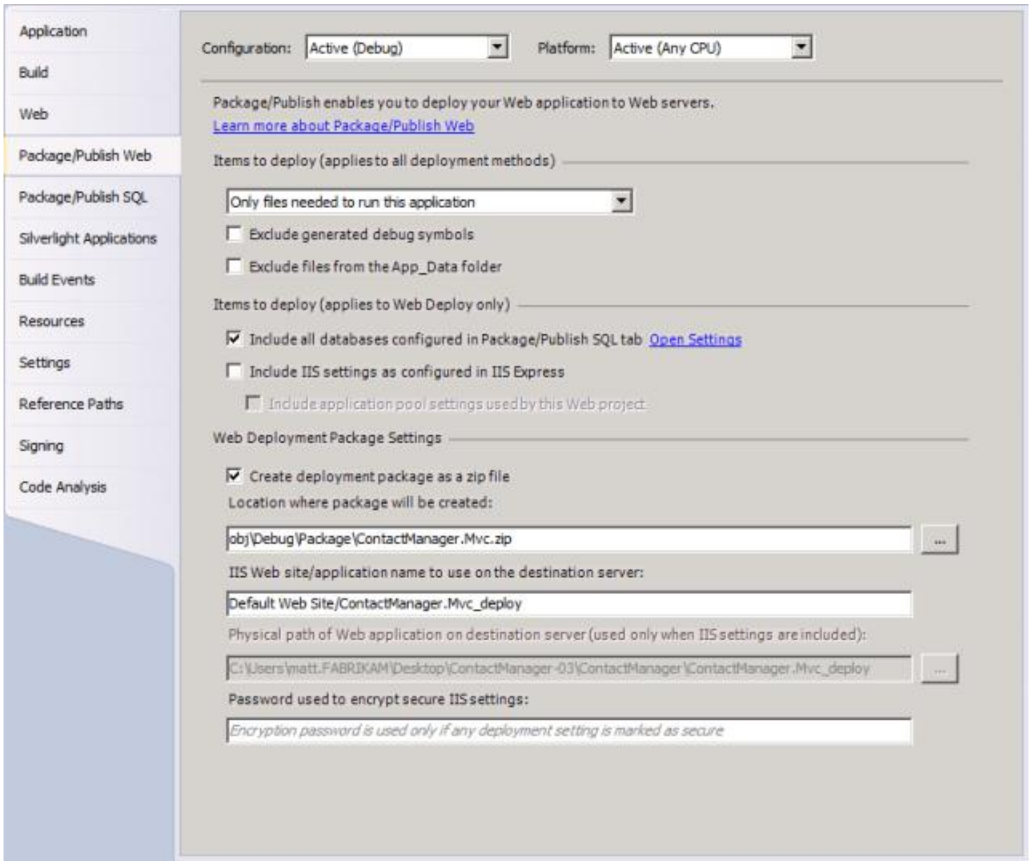
Project File Naming Conventions

When you create your own project files, you can use any file extension you like. However, to make your solutions easier for others to understand, you should use these common conventions:

- Use the .proj extension when you create a project file that builds projects.
- Use the .targets extension when you create a reusable project file to import into other project files. Files with a .targets extension typically don't build anything themselves, they simply contain instructions that you can import into your .proj files.

Integration with Deployment Technologies

If you've worked with web application projects in Visual Studio 2010, like ASP.NET web applications and ASP.NET MVC web applications, you'll know that these projects include built-in support for packaging and deploying the web application to a target environment. The **Properties** pages for these projects include **Package/Publish Web** and **Package/Publish SQL** tabs that you can use to configure how the components of your application are packaged and deployed. This shows the **Package/Publish Web** tab:



Configuration: Active (Debug)   Platform: Active (Any CPU)

Package/Publish enables you to deploy your Web application to Web servers.
Learn more about Package/Publish Web

Items to deploy (applies to all deployment methods)

Only files needed to run this application

☐ Exclude generated debug symbols

☐ Exclude files from the App_Data folder

Items to deploy (applies to Web Deploy only)

☑ Include all databases configured in Package/Publish SQL tab Open Settings

☐ Include IIS settings as configured in IIS Express

☐ Include application pool settings used by this Web project

Web Deployment Package Settings

☑ Create deployment package as a zip file

Location where package will be created:
obj\Debug\Package\ContactManager.Mvc.zip

IIS Web site/application name to use on the destination server:
Default Web Site/ContactManager.Mvc_deploy

Physical path of Web application on destination server (used only when IIS settings are included):
C:\Users\matt.FABRIKAM\Desktop\ContactManager-03\ContactManager\ContactManager.Mvc_deploy

Password used to encrypt secure IIS settings:
Encryption password is used only if any deployment setting is marked as secure

The underlying technology behind these capabilities is known as the Web Publishing Pipeline (WPP). The WPP essentially brings MSBuild and Web Deploy together to provide a complete build, package, and deployment process for your web applications.

The good news is that you can take advantage of the integration points that the WPP provides when you create custom project files for web projects. You can include deployment instructions in your project file, which allows you to build your projects, create web deployment packages, and install these packages on remote servers through a single project file and a single call to MSBuild. You can also call any other executables as part of your build process. For example, you can run the VSDBCMD.exe command-line tool to deploy a database from a schema file. Over the course of this topic, you'll see how you can take advantage of these capabilities to meet the requirements of your enterprise deployment scenarios.

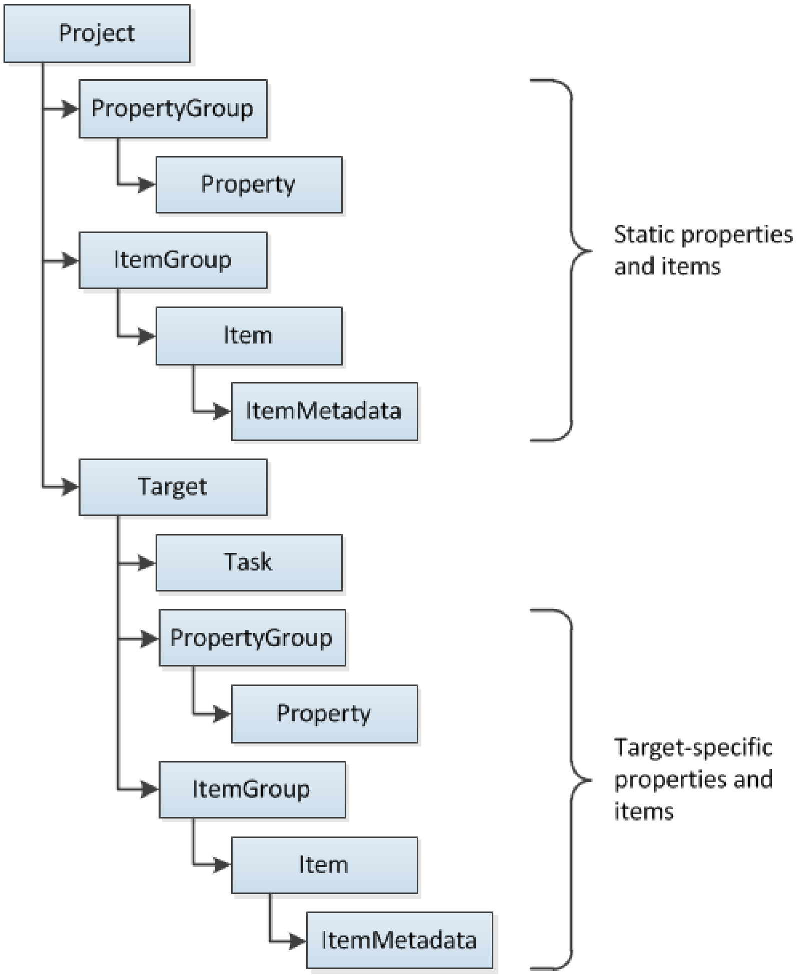
Note: For more information on how the web application deployment process works, see ASP.NET Web Application Project Deployment Overview.

The Anatomy of a Project File

Before you look at the build process in more detail, it's worth taking a few moments to familiarize yourself with the basic structure of an MSBuild project file. This section provides an overview of the more common elements that you'll encounter when you review, edit, or create a project file. In particular, you'll learn:

- How to use *properties* to manage variables for the build process.
- How to use *items* to identify the inputs to the build process, like code files.
- How to use *targets* and *tasks* to provide execution instructions to MSBuild, using *properties* and *items* defined elsewhere in the project file.

This shows the relationship between the key elements in an MSBuild project file:



The Project Element

The Project element is the root element of every project file. In addition to identifying the XML schema for the project file, the **Project** element can include attributes to specify the entry points for the build process. For example, in the Contact Manager sample solution, the *Publish.proj* file specifies that the build should start by calling the target named **FullPublish**.

XML

```
<Project ToolsVersion="4.0" DefaultTargets="FullPublish"
         xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

</Project>
```

Properties and Conditions

A project file typically needs to provide lots of different pieces of information in order to successfully build and deploy your projects. These pieces of information could include server names, connection strings, credentials, build configurations, source and destination file paths, and any other information you want to include to support customization. In a project file, properties must be defined within a PropertyGroup element. MSBuild properties consist of key-value pairs. Within the **PropertyGroup** element, the element name defines the property key and the content of the element defines the property value. For example, you could define properties named **ServerName** and **ConnectionString** to store a static server name and connection string.

XML

```
<PropertyGroup>
   <ServerName>FABRIKAM\TEST1</ServerName>
   <ConnectionString>
     Data Source=FABRIKAM\TESTDB;InitialCatalog=ContactManager,...
   </ConnectionString>
</PropertyGroup>
```

To retrieve a property value, you use the format **$(***PropertyName***)**. For example, to retrieve the value of the **ServerName** property, you would type:

```
$(ServerName)
```

Note: You'll see examples of how and when to use property values later in this topic.

Embedding information as static properties in a project file is not always the ideal approach to managing the build process. In a lot of scenarios, you'll want to obtain the information from other sources or empower the user to provide the information from the command prompt. MSBuild allows you to specify any property value as a command-line parameter. For example, the user could provide a value for **ServerName** when he or she runs MSBuild.exe from the command line.

```
msbuild.exe Publish.proj /p:ServerName=FABRIKAM\TESTWEB1
```

Note: For more information on the arguments and switches you can use with MSBuild.exe, see MSBuild Command Line Reference.

You can use the same property syntax to obtain the values of environment variables and built-in project properties. Lots of commonly used properties are defined for you, and you can use them in your project files by including the relevant parameter name. For example, to retrieve the current project platform—for example, **x86** or **AnyCpu**—you can include the **$(Platform)** property reference in your project file. For more information, see Macros for Build Commands and Properties, Common MSBuild Project Properties, and Reserved Properties.

Properties are often used in conjunction with *conditions*. Most MSBuild elements support the **Condition** attribute, which lets you specify the criteria upon which MSBuild should evaluate the element. For example, consider this property definition:

XML

```xml
<PropertyGroup>
   <OutputRoot Condition=" '$(OutputRoot)'=='' ">..\Publish\Out\</OutputRoot>
   ...
</PropertyGroup>
```

When MSBuild processes this property definition, it first checks to see whether an **$(OutputRoot)** property value is available. If the property value is blank—in other words, the user hasn't provided a value for this property—the condition evaluates to **true** and the property value is set to **..\Publish\Out**. If the user has provided a value for this property, the condition evaluates to **false** and the static property value is not used.

For more information on the different ways in which you can specify conditions, see MSBuild Conditions.

Items and Item Groups

One of the important roles of the project file is to define the inputs to the build process. Typically, these inputs are files—code files, configuration files, command files, and any other files that you need to process or copy as part of the build process. In the MSBuild project schema, these inputs are represented by Item elements. In a project file, items must be defined within an ItemGroup element. Just like **Property** elements, you can name an **Item** element however you like. However, you must specify an **Include** attribute to identify the file or wildcard that the item represents.

XML

```xml
<ItemGroup>
   <ProjectsToBuild Include="$(SourceRoot)ContactManager-WCF.sln"/>
</ItemGroup>
```

By specifying multiple **Item** elements with the same name, you're effectively creating a named list of resources. A good way to see this in action is to take a look inside one of the project files that Visual Studio creates. For example, the *ContactManager.Mvc.csproj* file in the sample solution includes a lot of item groups, each with several identically named **Item** elements.

XML

```
<ItemGroup>
   <Reference Include="Microsoft.CSharp" />
   <Reference Include="System.Runtime.Serialization" />
   <Reference Include="System.ServiceModel" />
   ...
</ItemGroup>
<ItemGroup>
   <Compile Include="Controllers\AccountController.cs" />
   <Compile Include="Controllers\ContactsController.cs" />
   <Compile Include="Controllers\HomeController.cs" />
   ...
</ItemGroup>
<ItemGroup>
   <Content Include="Content\Custom.css" />
   <Content Include="CreateDatabase.sql" />
   <Content Include="DropDatabase.sql" />
   ...
</ItemGroup>
```

In this way, the project file is instructing MSBuild to construct lists of files that need to be processed in the same way—the **Reference** list includes assemblies that must be in place for a successful build, the **Compile** list includes code files that must be compiled, and the **Content** list includes resources that must be copied unaltered. We'll look at how the build process references and uses these items later in this topic.

Item elements can also include ItemMetadata child elements. These are user-defined key-value pairs and essentially represent properties that are specific to that item. For example, a lot of the **Compile** item elements in the project file include **DependentUpon** child elements.

XML

```
<Compile Include="Global.asax.cs">
   <DependentUpon>Global.asax</DependentUpon>
</Compile>
```

Note: In addition to user-created item metadata, all items are assigned various common metadata on creation. For more information, see Well-known Item Metadata.

You can create **ItemGroup** elements within the root-level **Project** element or within specific **Target** elements. **ItemGroup** elements also support **Condition** attributes, which lets you tailor the inputs to the build process according to conditions like the project configuration or platform.

Targets and Tasks

In the MSBuild schema, a Task element represents an individual build instruction (or task). MSBuild includes a multitude of predefined tasks. For example:

- The **Copy** task copies files to a new location.

- The **Csc** task invokes the Visual C# compiler.
- The **Vbc** task invokes the Visual Basic compiler.
- The **Exec** task runs a specified program.
- The **Message** task writes a message to a logger.

> **Note:** For full details of the tasks that are available out of the box, see MSBuild Task Reference. For more information on tasks, including how to create your own custom tasks, see MSBuild Tasks.

Tasks must always be contained within Target elements. A **Target** element is a set of one or more tasks that are executed sequentially, and a project file can contain multiple targets. When you want to run a task, or a set of tasks, you invoke the target that contains them. For example, suppose you have a simple project file that logs a message.

XML

```xml
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
   <Target Name="LogMessage">
      <Message Text="Hello world!" />
   </Target>
</Project>
```

You can invoke the target from the command line, by using the **/t** switch to specify the target.

```
 msbuild.exe Publish.proj /t:LogMessage
```

Alternatively, you can add a **DefaultTargets** attribute to the **Project** element, to specify the targets that you want to invoke.

XML

```xml
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003"
         DefaultTargets="FullPublish">
   <Target Name="LogMessage">
      <Message Text="Hello world!" />
   </Target>
</Project>
```

In this case, you don't need to specify the target from the command line. You can simply specify the project file, and MSBuild will invoke the **FullPublish** target for you.

```
msbuild.exe Publish.proj
```

Both targets and tasks can include **Condition** attributes. As such, you can choose to omit entire targets or individual tasks if certain conditions are met.

Generally speaking, when you create useful tasks and targets, you'll need to refer to the properties and items that you've defined elsewhere in the project file:

- To use a property value, type **$(***PropertyName***)**, where *PropertyName* is the name of the **Property** element or the name of the parameter.
- To use an item, type **@(***ItemName***)**, where *ItemName* is the name of the **Item** element.

Note: Remember that if you create multiple items with the same name, you're building a list. In contrast, if you create multiple properties with the same name, the last property value you provide will overwrite any previous properties with the same name—a property can only contain a single value.

For example, in the *Publish.proj* file in the sample solution, take a look at the **BuildProjects** target.

XML

```
<Target Name="BuildProjects" Condition=" '$(BuildingInTeamBuild)'!='true' ">
   <MSBuild Projects="@(ProjectsToBuild)"
            Properties="OutDir=$(OutputRoot);
                        Configuration=$(Configuration);
                        DeployOnBuild=true;
                        DeployTarget=Package"
            Targets="Build" />
</Target>
```

In this sample, you can observe these key points:

- If the **BuildingInTeamBuild** parameter is specified and has a value of **true**, none of the tasks within this target will be executed.
- The target contains a single instance of the MSBuild task. This task lets you build other MSBuild projects.
- The **ProjectsToBuild** item is passed to the task. This item could represent a list of project or solution files, all defined by **ProjectsToBuild** item elements within an item group. In this case, the **ProjectsToBuild** item refers to a single solution file.

  XML

  ```
  <ItemGroup>
     <ProjectsToBuild Include="$(SourceRoot)ContactManager-WCF.sln"/>
  </ItemGroup>
  ```

- The property values passed to the **MSBuild** task include parameters named **OutputRoot** and **Configuration**. These are set to parameter values if they are provided, or static property values if they are not.

  XML

  ```
  <PropertyGroup>
     ...
     <Configuration Condition=" '$(Configuration)'=='' ">Release
     </Configuration>
     <OutputRoot Condition=" '$(OutputRoot)'=='' ">..\Publish\Out\
     </OutputRoot>
  ```

```
  ...
</PropertyGroup>
```

You can also see that the **MSBuild** task invokes a target named **Build**. This is one of several built-in targets that are widely used in Visual Studio project files and are available to you in your custom project files, like **Build**, **Clean**, **Rebuild**, and **Publish**. You'll learn more about using targets and tasks to control the build process, and about the **MSBuild** task in particular, later in this topic.

> **Note:** For more information on targets, see MSBuild Targets.

Splitting Project Files to Support Multiple Environments

Suppose you want to be able to deploy a solution to multiple environments, like test servers, staging platforms, and production environments. The configuration may vary substantially between these environments—not just in terms of server names, connection strings, and so on, but also potentially in terms of credentials, security settings, and lots of other factors. If you need to do this regularly, it's not really expedient to edit multiple properties in your project file every time you switch the target environment. Nor is it an ideal solution to require an endless list of property values to be provided to the build process.

Fortunately there is an alternative. MSBuild lets you split your build configuration across multiple project files. To see how this works, in the sample solution, notice that there are two custom project files:

- *Publish.proj*, which contains properties, items, and targets that are common to all environments.
- *Env-Dev.proj*, which contains properties that are specific to a developer environment.

Now notice that the *Publish.proj* file includes an Import element, immediately beneath the opening **Project** tag.

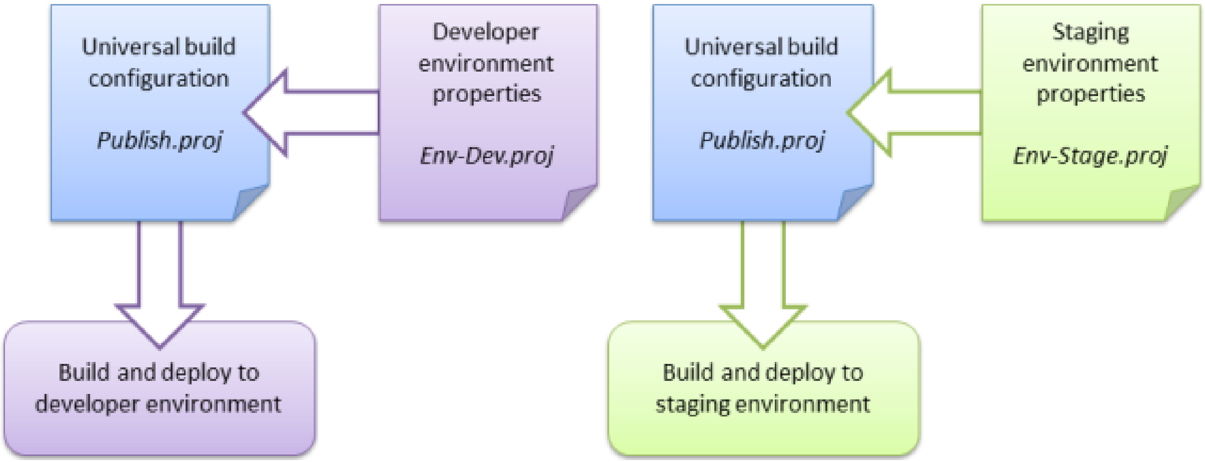
XML

```xml
<Import Project="$(TargetEnvPropsFile)"/>
```

The **Import** element is used to import the contents of another MSBuild project file into the current MSBuild project file. In this case, the **TargetEnvPropsFile** parameter provides the filename of the project file you want to import. You can provide a value for this parameter when you run MSBuild.

```
msbuild.exe Publish.proj /p:TargetEnvPropsFile=EnvConfig\Env-Dev.proj
```

This effectively merges the contents of the two files into a single project file. Using this approach, you can create one project file containing your universal build configuration and multiple supplementary project files containing environment-specific properties. As a result, simply running a command with a different parameter value lets you deploy your solution to a different environment.


Universal build configuration
Publish.proj
Developer environment properties
Env-Dev.proj
Build and deploy to developer environment
Universal build configuration
Publish.proj
Staging environment properties
Env-Stage.proj
Build and deploy to staging environment

Splitting your project files in this way is a good practice to follow. It allows developers to deploy to multiple environments by running a single command, while avoiding the duplication of universal build properties across multiple project files.

Note: For guidance on how to customize the environment-specific project files for your own server environments, see Configure Deployment Properties for a Target Environment.

Conclusion

This topic provided a general introduction to MSBuild project files and explained how you can create your own custom project files to control the build process. It also introduced the concept of splitting project files into universal build instructions and environment-specific build properties, to make it easy to build and deploy projects to multiple destinations.

The next topic, Understanding the Build Process, provides more insight into how you can use project files to control build and deployment by walking you through the deployment of a solution with a realistic level of complexity.

Further Reading

For a more in-depth introduction to project files and the WPP, see Inside the Microsoft Build Engine: Using MSBuild and Team Foundation Build by Sayed Ibrahim Hashimi and William Bartholomew, ISBN: 978-0-7356-4524-0.

Understanding the Build Process

This topic provides a walkthrough of an enterprise-scale build and deployment process. The approach described in this topic uses custom Microsoft Build Engine (MSBuild) project files to provide fine-grained control over every aspect of the process. Within the project files, custom MSBuild targets are used to run deployment utilities like the Internet Information Services (IIS) Web Deployment Tool (MSDeploy.exe) and the database deployment utility VSDBCMD.exe.

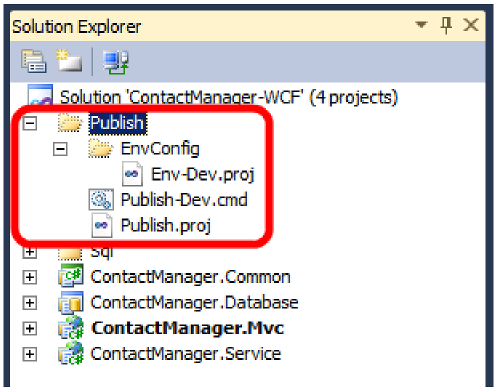
Note: The previous topic, Understanding the Project File, described the key components of an MSBuild project file and introduced the concept of split project files to support deployment to multiple target environments. If you're not already familiar with these concepts, you should review Understanding the Project File before you work through this topic.

Build and Deployment Overview

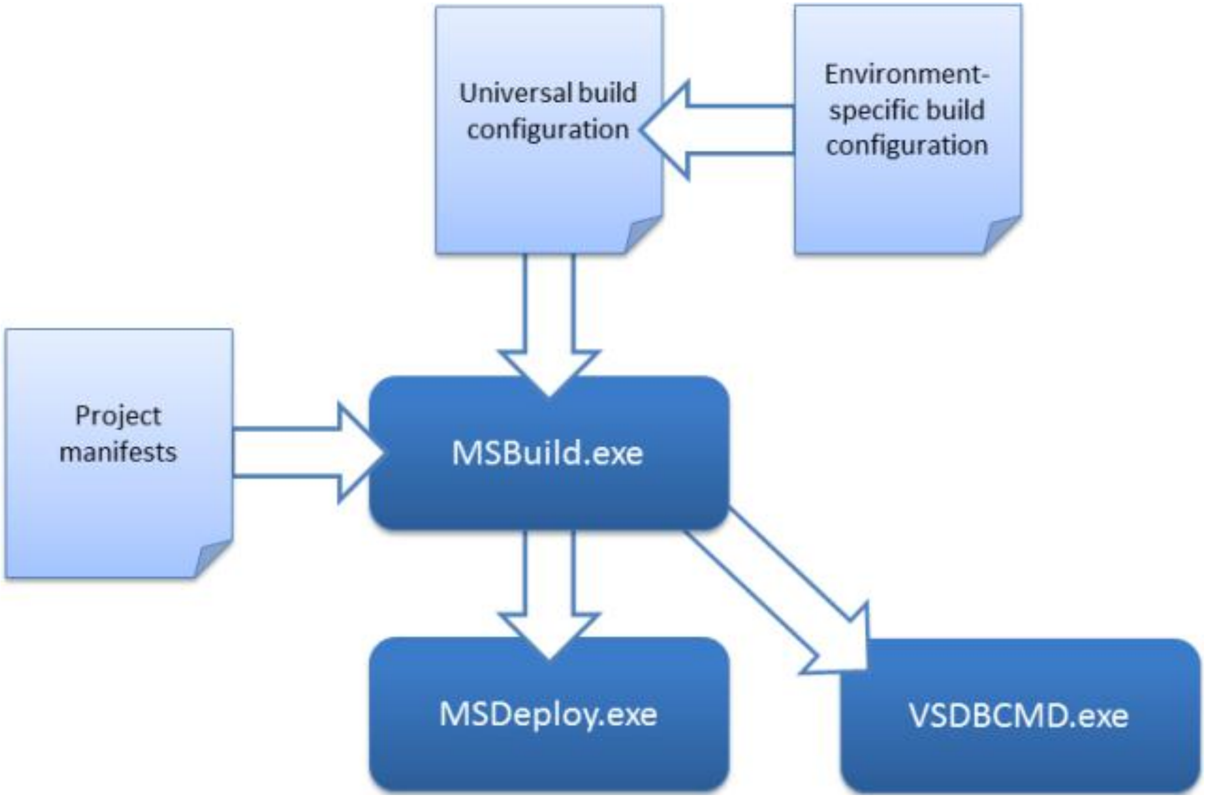
In the Contact Manager solution, three files control the build and deployment process:

- A *universal project file* (*Publish.proj*). This contains build and deployment instructions that do not change between destination environments.
- An *environment-specific project file* (*Env-Dev.proj*). This contains build and deployment settings that are specific to a particular destination environment. For example, you could use the *Env-Dev.proj* file to provide settings for a developer or test environment and create an alternative file named *Env-Stage.proj* to provide settings for a staging environment.
- A *command file* (*Publish-Dev.cmd*). This contains an MSBuild.exe command that specifies which project files you want to execute. You can create a command file for every destination environment, where each file contains an MSBuild.exe command that specifies a different environment-specific project file. This lets the developer deploy to different environments simply by running the appropriate command file.

In the sample solution, you can find these three files in the Publish solution folder.



Before you look at these files in more detail, let's take a look at how the overall build process works when you use this approach. At a high level, the build and deployment process looks like this:


Universal build configuration
Environment-specific build configuration
Project manifests
MSBuild.exe
MSDeploy.exe
VSDBCMD.exe

The first thing that happens is that the two project files—one containing universal build and deployment instructions, and one containing environment-specific settings—are merged into a single project file. MSBuild then works through the instructions in the project file. It builds each of the projects in the solution, using the project file for each project. It then calls out to other tools, like Web Deploy (MSDeploy.exe) and the VSDBCMD utility to deploy your web content and databases to the target environment.

From start to finish, the build and deployment process performs these tasks:

1. It deletes the contents of the output directory, in preparation for a fresh build.
2. It builds each project in the solution:
    a. For web projects—in this case, an ASP.NET MVC web application and a WCF web service—the build process creates a web deployment package for each project.
    b. For database projects, the build process creates a deployment manifest (.deploymanifest file) for each project.
3. It uses the VSDBCMD.exe utility to deploy each database project in the solution, using various properties from the project files—a target connection string and a database name—together with the .deploymanifest file.
4. It uses the MSDeploy.exe utility to deploy each web project in the solution, using various properties from the project files to control the deployment process.

You can use the sample solution to trace this process in more detail.

> **Note:** For guidance on how to customize the environment-specific project files for your own server environments, see Configure Deployment Properties for a Target Environment.

Invoking the Build and Deployment Process

To deploy the Contact Manager solution to a developer test environment, the developer runs the *Publish-Dev.cmd* command file. This invokes MSBuild.exe, specifying *Publish.proj* as the project file to execute and *Env-Dev.proj* as a parameter value.

```
msbuild.exe Publish.proj /fl /p:TargetEnvPropsFile=EnvConfig\Env-Dev.proj
```

> **Note:** The **/fl** switch (short for **/fileLogger**) logs the build output to a file named *msbuild.log* in the current directory. For more information, see the MSBuild Command Line Reference.

At this point, MSBuild starts running, loads the *Publish.proj* file, and starts processing the instructions within it. The first instruction tells MSBuild to import the project file that the **TargetEnvPropsFile** parameter specifies.

XML

```xml
<Import Project="$(TargetEnvPropsFile)" />
```

The **TargetEnvPropsFile** parameter specifies the *Env-Dev.proj* file, so MSBuild merges the contents of the *Env-Dev.proj* file into the *Publish.proj* file.

The next elements that MSBuild encounters in the merged project file are property groups. Properties are processed in the order in which they appear in the file. MSBuild creates a key-value pair for each property, providing that any specified conditions are met. Properties defined later in the file will overwrite any properties with the same name defined earlier in the file. For example, consider the **OutputRoot** properties.

XML

```xml
<OutputRoot Condition=" '$(OutputRoot)'=='' ">..\Publish\Out\</OutputRoot>
<OutputRoot Condition=" '$(BuildingInTeamBuild)'=='true' ">$(OutDir)</OutputRoot>
```

When MSBuild processes the first **OutputRoot** element, providing a similarly named parameter has not been provided, it sets the value of the **OutputRoot** property to **..\Publish\Out**. When it encounters the second **OutputRoot** element, if the condition evaluates to **true**, it will overwrite the value of the **OutputRoot** property with the value of the **OutDir** parameter.

The next element that MSBuild encounters is a single item group, containing an item named **ProjectsToBuild**.

XML

```xml
<ItemGroup>
   <ProjectsToBuild Include="$(SourceRoot)ContactManager-WCF.sln"/>
</ItemGroup>
```

MSBuild processes this instruction by building an item list named **ProjectsToBuild**. In this case, the item list contains a single value—the path and filename of the solution file.

At this point, the remaining elements are targets. Targets are processed differently from properties and items—essentially, targets are not processed unless they are either explicitly specified by the user or invoked by another construct within the project file. Recall that the opening **Project** tag includes a **DefaultTargets** attribute.

XML

```xml
<Project ToolsVersion="4.0"
         DefaultTargets="FullPublish"
         xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
```

This instructs MSBuild to invoke the **FullPublish** target, if targets are not specified when MSBuild.exe is invoked. The **FullPublish** target doesn't contain any tasks; instead it simply specifies a list of dependencies.

XML

```xml
<PropertyGroup>
  <FullPublishDependsOn>
     Clean;
     BuildProjects;
     GatherPackagesForPublishing;
     PublishDbPackages;
     PublishWebPackages;
  </FullPublishDependsOn>
</PropertyGroup>
<Target Name="FullPublish" DependsOnTargets="$(FullPublishDependsOn)" />
```

This dependency tells MSBuild that in order to execute the **FullPublish** target, it needs to invoke this list of targets in the order provided:

1. It must invoke the **Clean** target.
2. It must invoke the **BuildProjects** target.
3. It must invoke the **GatherPackagesForPublishing** target.
4. It must invoke the **PublishDbPackages** target.
5. It must invoke the **PublishWebPackages** target.

The Clean Target

The **Clean** target basically deletes the output directory and all its contents, as preparation for a fresh build.

XML

```xml
<Target Name="Clean" Condition=" '$(BuildingInTeamBuild)'!='true' ">
  <Message Text="Cleaning up the output directory [$(OutputRoot)]"/>
  <ItemGroup>
     <_FilesToDelete Include="$(OutputRoot)**\*"/>
  </ItemGroup>
  <Delete Files="@(_FilesToDelete)"/>
  <RemoveDir Directories="$(OutputRoot)"/>
</Target>
```

Notice that the target includes an **ItemGroup** element. When you define properties or items within a **Target** element, you’re creating *dynamic* properties and items. In other words, the properties or items aren't processed until the target is executed. The output directory might not exist or contain any files until the build process begins, so you can't build the **_FilesToDelete** list as a static item; you have to wait until execution is underway. As such, you build the list as a dynamic item within the target.

> **Note:** In this case, because the **Clean** target is the first to be executed, there's no real need to use a dynamic item group. However, it's good practice to use dynamic properties and items in this type of scenario, as you might want to execute targets in a different order at some point.
>
> You should also aim to avoid declaring items that will never be used. If you have items that will only be used by a specific target, consider placing them inside the target to remove any unnecessary overhead on the build process.

Dynamic items aside, the **Clean** target is fairly straightforward and makes use of the built-in **Message**, **Delete**, and **RemoveDir** tasks to:

1. Send a message to the logger.
2. Build a list of files to delete.
3. Delete the files.
4. Remove the output directory.

The BuildProjects Target

The **BuildProjects** target basically builds all the projects in the sample solution.

XML

```
<Target Name="BuildProjects" Condition=" '$(BuildingInTeamBuild)'!='true' ">
   <MSBuild Projects="@(ProjectsToBuild)"
            Properties="OutDir=$(OutputRoot);
                        Configuration=$(Configuration);
                        DeployOnBuild=true;
                        DeployTarget=Package"
            Targets="Build" />
  </Target>
```

This target was described in some detail in the previous topic, Understanding the Project File, to illustrate how tasks and targets reference properties and items. At this point, you're mainly interested in the **MSBuild** task. You can use this task to build multiple projects. The task does not create a new instance of MSBuild.exe; it uses the current running instance to build each project. The key points of interest in this example are the deployment properties:

- The **DeployOnBuild** property instructs MSBuild to run any deployment instructions in the project settings when the build of each project is complete.
- The **DeployTarget** property identifies the target that you want to invoke after the project is built. In this case, the **Package** target builds the project output into a deployable web package.

> **Note:** The **Package** target invokes the Web Publishing Pipeline (WPP), which provides integration between MSBuild and Web Deploy. If you want to take a look at the built-in targets that the WPP provides, review the *Microsoft.Web.Publishing.targets* file in the %PROGRAMFILES(x86)%\MSBuild\Microsoft\VisualStudio\v10.0\Web folder.

The GatherPackagesForPublishing Target

If you study the **GatherPackagesForPublishing** target, you'll notice that it doesn't actually contain any tasks. Instead, it contains a single item group that defines three dynamic items.

XML

```xml
<Target Name="GatherPackagesForPublishing">
   <ItemGroup>
      <PublishPackages 
         Include="$(_ContactManagerDest)ContactManager.Mvc.deploy.cmd">
         <WebPackage>true</WebPackage>
         <!-- More item metadata -->
      </PublishPackages>
      <PublishPackages 
         Include="$(_ContactManagerSvcDest)ContactManager.Service.deploy.cmd">
         <WebPackage>true</WebPackage>
         <!-- More item metadata -->
      </PublishPackages>
      <DbPublishPackages Include="$(_DbDeployManifestPath)">
         <DbPackage>true</DbPackage>
         <!-- More item metadata -->
      </DbPublishPackages>
   </ItemGroup>
</Target>
```

These items refer to the deployment packages that were created when the **BuildProjects** target was executed. You couldn't define these items statically in the project file, because the files to which the items refer don't exist until the **BuildProjects** target is executed. Instead, the items must be defined dynamically within a target that is not invoked until after the **BuildProjects** target is executed.

The items are not used within this target—this target simply builds the items and the metadata associated with each item value. Once these elements are processed, the **PublishPackages** item will contain two values, the path to the *ContactManager.Mvc.deploy.cmd* file and the path to the

ContactManager.Service.deploy.cmd file. Web Deploy creates these files as part of the web package for each project, and these are the files that you must invoke on the destination server in order to deploy the packages. If you open up one of these files, you'll basically see an MSDeploy.exe command with various build-specific parameter values.

The **DbPublishPackages** item will contain a single value, the path to the *ContactManager.Database.deploymanifest* file.

> **Note:** A .deploymanifest file is generated when you build a database project, and it uses the same schema as an MSBuild project file. It contains all the information required to deploy a database, including the location of the database schema (.dbschema) and details of any pre-deployment and post-deployment scripts. For more information, see An Overview of Database Build and Deployment.

You'll learn more about how deployment packages and database deployment manifests are created and used in Building and Packaging Web Application Projects and Deploying Database Projects.

The PublishDbPackages Target

Briefly speaking, the **PublishDbPackages** target invokes the VSDBCMD utility to deploy the **ContactManager** database to a target environment. Configuring database deployment involves lots of decisions and nuances, and you'll learn more about this in Deploying Database Projects and Customizing Database Deployments for Multiple Environments. In this topic, we'll focus on how this target actually functions.

First, notice that the opening tag includes an **Outputs** attribute.

XML

```xml
<Target Name="PublishDbPackages" Outputs="%(DbPublishPackages.Identity)">
```

This is an example of *target batching*. In MSBuild project files, batching is a technique for iterating over collections. The value of the **Outputs** attribute, **"%(DbPublishPackages.Identity)"**, refers to the **Identity** metadata property of the **DbPublishPackages** item list. This notation, **Outputs=%***(ItemList.ItemMetadataName)*, is translated as:

- Split the items in **DbPublishPackages** into batches of items that contain the same **Identity** metadata value.
- Execute the target once per batch.

> Note: **Identity** is one of the built-in metadata values that is assigned to every item on creation. It refers to the value of the **Include** attribute in the **Item** element—in other words, the path and filename of the item.

In this case, because there should never be more than one item with the same path and filename, we're essentially working with batch sizes of one. The target is executed once for every database package.

You can see a similar notation in the **_Cmd** property, which builds a VSDBCMD command with the appropriate switches.

XML

```xml
<_Cmd>"$(VsdbCmdExe)"
   /a:Deploy
   /cs:"%(DbPublishPackages.DatabaseConnectionString)"
   /p:TargetDatabase=%(DbPublishPackages.TargetDatabase)
   /manifest:"%(DbPublishPackages.FullPath)"
   /script:"$(_CmDbScriptPath)"
   $(_DbDeployOrScript)
</_Cmd>
```

In this case, **%(DbPublishPackages.DatabaseConnectionString)**, **%(DbPublishPackages.TargetDatabase)**, and **%(DbPublishPackages.FullPath)** all refer to metadata values of the **DbPublishPackages** item collection. The **_Cmd** property is used by the **Exec** task, which invokes the command.

XML

```xml
<Exec Command="$(_Cmd)"/>
```

As a result of this notation, the **Exec** task will create batches based on unique combinations of the **DatabaseConnectionString**, **TargetDatabase**, and **FullPath** metadata values, and the task will execute once for each batch. This is an example of *task batching*. However, because the target-level batching has already divided our item collection into single-item batches, the **Exec** task will run once and only once for each iteration of the target. In other words, this task invokes the VSDBCMD utility once for each database package in the solution.

> **Note:** For more information on target and task batching, see MSBuild Batching, Item Metadata in Target Batching, and Item Metadata in Task Batching.

The PublishWebPackages Target

By this point, you've invoked the **BuildProjects** target, which generates a web deployment package for each project in the sample solution. Accompanying each package is a *deploy.cmd* file, which contains the MSDeploy.exe commands required to deploy the package to the target environment, and a *SetParameters.xml* file, which specifies the necessary details of the target environment. You've also invoked the **GatherPackagesForPublishing** target, which generates an item collection containing the *deploy.cmd* files you're interested in. Essentially, the **PublishWebPackages** target performs these functions:

- It manipulates the *SetParameters.xml* file for each package to include the correct details for the target environment, using the **XmlPoke** task.
- It invokes the *deploy.cmd* file for each package, using the appropriate switches.

Just like the **PublishDbPackages** target, the **PublishWebPackages** target uses target batching to ensure that the target is executed once for each web package.

XML

```xml
<Target Name="PublishWebPackages" Outputs="%(PublishPackages.Identity)">
```

Within the target, the **Exec** task is used to run the *deploy.cmd* file for each web package.

XML

```xml
<PropertyGroup>
   <_Cmd>
      %(PublishPackages.FullPath)
      $(_WhatifSwitch)
      /M:$(MSDeployComputerName)
      %(PublishPackages.AdditionalMSDeployParameters)
   </_Cmd>
</PropertyGroup>
<Exec Command="$(_Cmd)"/>
```

For more information on configuring the deployment of web packages, see Building and Packaging Web Application Projects.

Conclusion

This topic provided a walkthrough of how split project files are used to control the build and deployment process from start to finish for the Contact Manager sample solution. Using this approach lets you run complex, enterprise-scale deployments in a single, repeatable step, simply by running an environment-specific command file.

Further Reading

For a more in-depth introduction to project files and the WPP, see Inside the Microsoft Build Engine: Using MSBuild and Team Foundation Build by Sayed Ibrahim Hashimi and William Bartholomew, ISBN: 978-0-7356-4524-0.

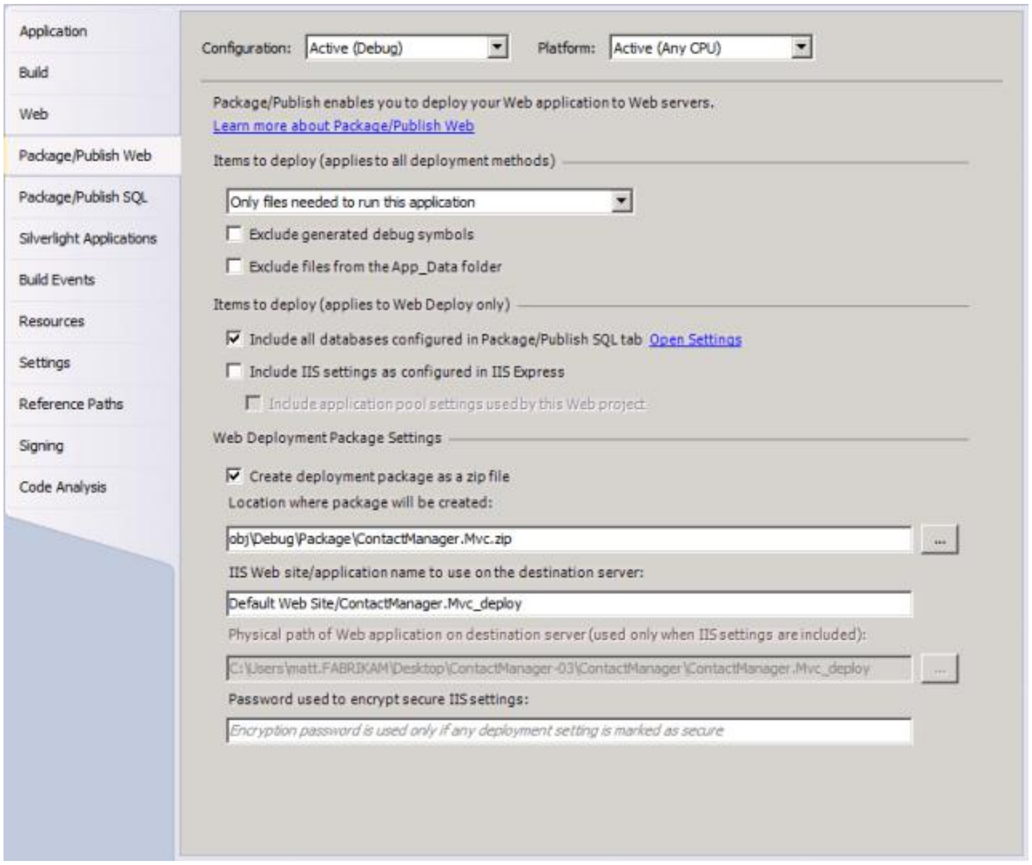
Building and Packaging Web Application Projects

When you want to deploy a web application project to a remote server environment, your first task is to build the project and generate a web deployment package. This topic describes how the build process works for web application projects. In particular, it explains:

- How the Web Publishing Pipeline (WPP) extends the build process to include deployment functionality.
- How the Internet Information Services (IIS) Web Deployment Tool (Web Deploy) turns your web application into a deployment package.
- How the build and packaging process works and what files are created.

Web Application Projects and the WPP

In Visual Studio 2010, the build and deployment process for web application projects is supported by the WPP. The WPP provides a set of Microsoft Build Engine (MSBuild) targets that extend the functionality of MSBuild and enable it to integrate with Web Deploy. Within Visual Studio, you can see this extended functionality on the property pages for your web application project. The **Package/Publish Web** page, together with the **Package/Publish SQL** page, lets you configure how your web application project is packaged for deployment when the build process is complete.



Configuration: Active (Debug) Platform: Active (Any CPU)

Package/Publish enables you to deploy your Web application to Web servers.
Learn more about Package/Publish Web

Items to deploy (applies to all deployment methods)

Only files needed to run this application

☐ Exclude generated debug symbols

☐ Exclude files from the App_Data folder

Items to deploy (applies to Web Deploy only)

☑ Include all databases configured in Package/Publish SQL tab Open Settings

☐ Include IIS settings as configured in IIS Express

☐ Include application pool settings used by this Web project

Web Deployment Package Settings

☑ Create deployment package as a zip file

Location where package will be created:

obj\Debug\Package\ContactManager.Mvc.zip

IIS Web site/application name to use on the destination server:

Default Web Site/ContactManager.Mvc_deploy

Physical path of Web application on destination server (used only when IIS settings are included):

C:\Users\matt.FABRIKAM\Desktop\ContactManager-03\ContactManager\ContactManager.Mvc_deploy

Password used to encrypt secure IIS settings:

Encryption password is used only if any deployment setting is marked as secure

How Does the WPP Work?

If you take a look at the project file for a C#-based web application project, you can see that it imports two .targets files.

XML

```xml
<Import Project="$(MSBuildBinPath)\Microsoft.CSharp.targets" />
<Import Project="$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\
                 v10.0\WebApplications\Microsoft.WebApplication.targets" />
```

The first **Import** statement is common to all Visual C# projects. This file, *Microsoft.CSharp.targets*, contains targets and tasks that are specific to Visual C#. For example, the C# compiler (**Csc**) task is invoked here. The *Microsoft.CSharp.targets* file in turn imports the *Microsoft.Common.targets* file. This defines targets that are common to all projects, like **Build**, **Rebuild**, **Run**, **Compile**, and **Clean**. The second **Import** statement is specific to web application projects. The *Microsoft.WebApplication.targets* file in turn imports the *Microsoft.Web.Publishing.targets* file. The *Microsoft.Web.Publishing.targets* file essentially *is* the WPP. It defines targets, like **Package** and **MSDeployPublish**, that invoke Web Deploy to complete various deployment tasks.

To understand how these additional targets are used, in the Contact Manager sample solution, open the *Publish.proj* file and take a look at the **BuildProjects** target.

XML

```xml
<Target Name="BuildProjects" Condition=" '$(BuildingInTeamBuild)'!='true' ">
  <MSBuild Projects="@(ProjectsToBuild)"
           Properties="OutDir=$(OutputRoot);
                       Configuration=$(Configuration);
                       DeployOnBuild=true;
                       DeployTarget=Package"
           Targets="Build" />
</Target>
```

This target uses the **MSBuild** task to build various projects. Notice the **DeployOnBuild** and **DeployTarget** properties:

- The **DeployOnBuild=true** property essentially means "I want to execute an additional target when build completes successfully."
- The **DeployTarget** property identifies the name of the target you want to execute when the **DeployOnBuild** property is equal to **true**. In this case, you're specifying that you want MSBuild to execute the **Package** target after building the project.

The **Package** target is defined in the *Microsoft.Web.Publishing.targets* file. Essentially, this target takes the build output of your web application project and turns it into a web deployment package that can be published to an IIS web server.

> **Note:** To view a project file (for example, *ContactManager.Mvc.csproj*) in Visual Studio 2010, you first need to unload the project from your solution. In the **Solution Explorer** window, right-click the project node, and then click **Unload Project**. Right-click the project node again, and then click **Edit** *[project file]*). The project file will open in its raw XML form. Remember to reload the project when you're done.
>
> For more information on MSBuild targets, tasks, and **Import** statements, see Understanding the Project File. For a more in-depth introduction to project files and the WPP, see Inside the Microsoft Build Engine: Using MSBuild and Team Foundation Build by Sayed Ibrahim Hashimi and William Bartholomew, ISBN: 978-0-7356-4524-0.

What Is a Web Deployment Package?

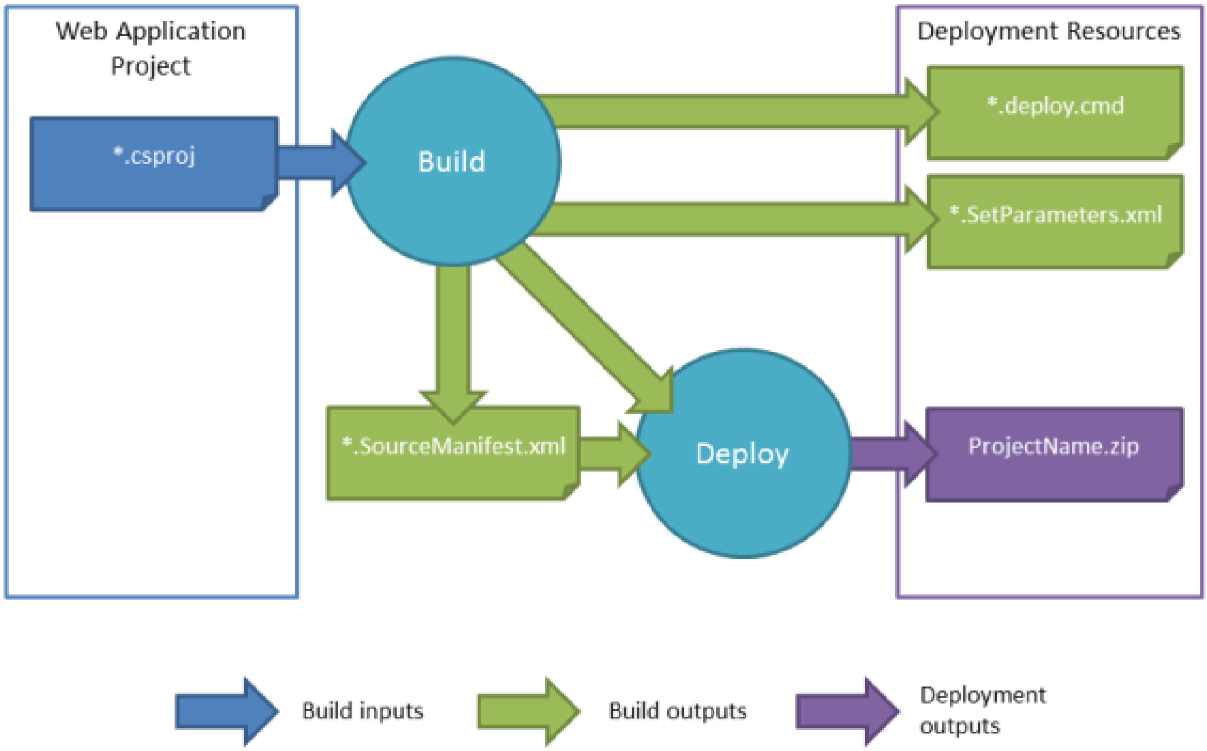
When you build and deploy a web application project, either by using Visual Studio 2010 or by using MSBuild directly, the end result is typically a *web deployment package*. The web deployment package is a .zip file. It contains everything that IIS and Web Deploy need in order to recreate your web application, including:

- The compiled output of your web application, including content, resource files, configuration files, JavaScript and cascading style sheets (CSS) resources, and so on.
- Assemblies for your web application project and for any referenced projects within your solution.
- SQL scripts to generate any databases that you're deploying with your web application.

Once the web deployment package has been generated, you can publish it to an IIS web server in various ways. For example, you can deploy it remotely by targeting the Web Deploy Remote Agent service or the Web Deploy Handler on the destination web server, or you can use IIS Manager to manually import the package on the destination web server. For more information on these approaches to deployment, see Choosing the Right Approach to Web Deployment.

How Does the Build Process Work?

This shows what happens when you build and package a web application project:



When you build a web application project, the build process generates a file named *[project name].SourceManifest.xml*. Along with the project file and the build output, this *.SourceManifest.xml* file tells Web Deploy what it needs to include in the web deployment package. Using these inputs, Web Deploy generates a web deployment package named *[project name].zip*.

Alongside the web deployment package, the build process generates two files that can help you to use the package:

- The *.deploy.cmd* file includes a set of parameterized Web Deploy (MSDeploy.exe) commands that publish your web deployment package to a remote IIS web server. Running the *.deploy.cmd* file, with appropriate parameters, typically provides a quicker and easier alternative to manually constructing the MSDeploy.exe commands yourself.
- The *SetParameters.xml* file provides a set of parameter values to the MSDeploy.exe command. These values include properties like the name of the IIS web application to which you want to deploy the package, the values of any service endpoints and connection strings defined in the *web.config* file, and any deployment property values defined on the project properties pages.

The *SetParameters.xml* file is key to managing the deployment process. This file is generated dynamically according to the contents of your web application project. For example, if you add a connection string to your *web.config* file, the build process will automatically detect the connection string, parameterize the deployment accordingly, and create an entry in the *SetParameters.xml* file to allow you to modify the connection string as part of the deployment process. The next topic, Configuring Parameters for Web Package Deployment, explains the role of this file in more detail and describes the different ways in which you can modify it during build and deployment.

Note: In Visual Studio 2010, the WPP does not support precompiling the pages in a web application prior to packaging. The next version of Visual Studio and the WPP will include the ability to precompile a web application as a packaging option.

Conclusion

This topic provided an overview of the build and packaging process for web application projects in Visual Studio 2010. It described how the WPP lets you invoke Web Deploy commands from MSBuild, and it explained how the build and packaging process works.

Once you've created a web deployment package, your next step is to deploy it. For more information on this, see Configuring Parameters for Web Package Deployment and Deploying Web Packages.

Further Reading

The next topics in this tutorial, Configuring Parameters for Web Package Deployment and Deploying Web Packages, provide guidance on how to use the web package you've created. The final tutorial in this series, Advanced Enterprise Web Deployment, provides guidance on how to customize and troubleshoot the packaging process.

For a more in-depth introduction to project files and the WPP, see Inside the Microsoft Build Engine: Using MSBuild and Team Foundation Build by Sayed Ibrahim Hashimi and William Bartholomew, ISBN: 978-0-7356-4524-0.

Configuring Parameters for Web Package Deployment

This topic describes how to set parameter values, like Internet Information Services (IIS) web application names, connection strings, and service endpoints, when you deploy a web package to a remote IIS web server.

Understanding Parameterization

When you build a web application project, the build and packaging process generates three key files:

- A *[project name].zip* file. This is the web deployment package for your web application project. This package contains all the assemblies, files, database scripts, and resources required to recreate your web application on a remote IIS web server.
- A *[project name].deploy.cmd* file. This contains a set of parameterized Web Deploy (MSDeploy.exe) commands that publish your web deployment package to a remote IIS web server.
- A *[project name].SetParameters.xml* file. This provides a set of parameter values to the MSDeploy.exe command. You can update the values in this file and pass it to Web Deploy as a command-line parameter when you deploy your web package.

Note: For more information on the build and packaging process, see Building and Packaging Web Application Projects.

The *SetParameters.xml* file is dynamically generated from your web application project file and any configuration files within your project. When you build and package your project, the Web Publishing Pipeline (WPP) will automatically detect lots of the variables that are likely to change between deployment environments, like the destination IIS web application and any database connection strings. These values are automatically parameterized in the web deployment package and added to the *SetParameters.xml* file. For example, if you add a connection string to the *web.config* file in your web application project, the build process will detect this change and will add an entry to the *SetParameters.xml* file accordingly.

In a lot of cases, this automatic parameterization will be sufficient. However, if your users need to vary other settings between deployment environments, like application settings or service endpoint URLs, you need to tell the WPP to parameterize these values in the deployment package and add corresponding entries to the *SetParameters.xml* file. The sections that follow explain how to do this.

Automatic Parameterization

When you build and package a web application, the WPP will automatically parameterize these things:

- The destination IIS web application path and name.
- Any connection strings in your *web.config* file.

- Connection strings for any databases you add to the **Package/Publish SQL** tab in the project property pages.

For example, if you were to build and package the Contact Manager sample solution without touching the parameterization process in any way, the WPP would generate this *ContactManager.Mvc.SetParameters.xml* file:

XML

```
<parameters>
  <setParameter
    name="IIS Web Application Name"
    value="Default Web Site/ContactManager.Mvc_deploy" />
  <setParameter
    name="ApplicationServices-Web.config Connection String"
    value="Data Source=DEVWORKSTATION\SQLEXPRESS;Initial Catalog=CMAppServices;
           Integrated Security=true;" />
</parameters>
```

In this case:

- The **IIS Web Application Name** parameter is the IIS path where you want to deploy the web application. The default value is taken from the **Package/Publish Web** page in the project property pages.
- The **ApplicationServices-Web.config Connection String** parameter was generated from a **connectionStrings/add** element in the *web.config* file. It represents the connection string that the application should use to contact the membership database. The value you provide here will be substituted into the deployed *web.config* file. The default value is taken from the pre-deployment *web.config* file.

The WPP also parameterizes these properties in the deployment package it generates. You can provide values for these properties when you install the deployment package. If you install the package manually through IIS Manager, as described in Manually Installing Web Packages, the installation wizard prompts you to provide values for any parameters. If you install the package remotely using the *.deploy.cmd* file, as described in Deploying Web Packages, Web Deploy will look to this *SetParameters.xml* file to provide the parameter values. You can edit the values in the *SetParameters.xml* file manually, or you can customize the file as part of an automated build and deployment process. This process is described in more detail later in this topic.

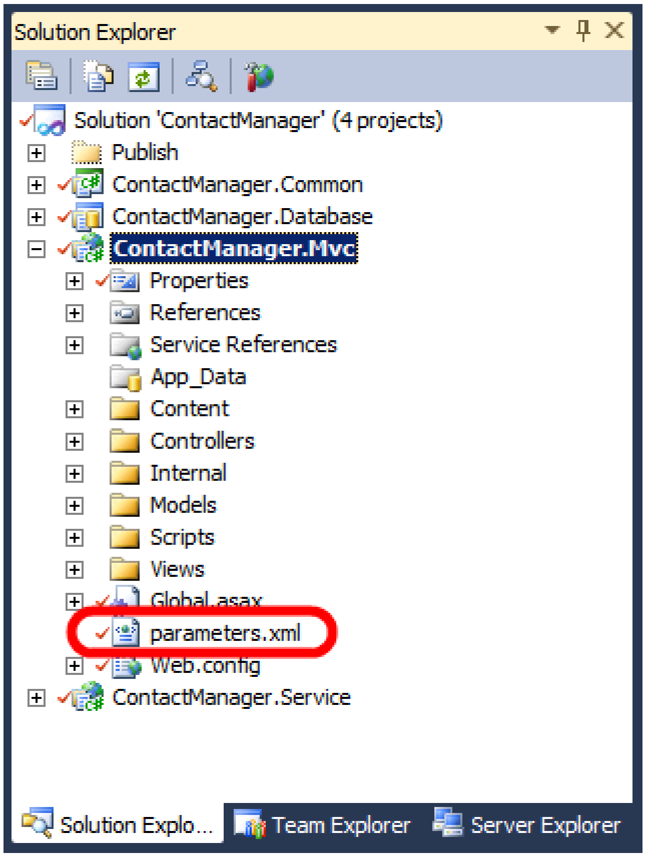
Custom Parameterization

In more complex deployment scenarios, you'll often want to parameterize additional properties before you deploy your project. Generally speaking, you should parameterize any properties and settings that will vary between destination environments. These can include:

- Service endpoints in the *web.config* file.

- Application settings in the *web.config* file.
- Any other declarative properties that you want to prompt users to specify.

The easiest way to parameterize these properties is to add a *parameters.xml* file to the root folder of your web application project. For example, in the Contact Manager solution, the ContactManager.Mvc project includes a *parameters.xml* file in the root folder.



If you open this file, you'll see that it contains a single **parameter** entry. The entry uses an XML Path Language (XPath) query to locate and parameterize the endpoint URL of the ContactService Windows Communication Foundation (WCF) service in the *web.config* file.

XML

```
<parameters>
  <parameter name="ContactService Service Endpoint Address"
             description="Specify the endpoint URL for the ContactService WCF
                          service in the destination environment"
             defaultValue="http://localhost/ContactManagerService">
    <parameterEntry kind="XmlFile" scope="Web.config"
                    match="/configuration/system.serviceModel/client
                           /endpoint[@name='BasicHttpBinding_IContactService']
                           /@address" />
  </parameter>
```

```xml
</parameters>
```

In addition to parameterizing the endpoint URL in the deployment package, the WPP also adds a corresponding entry to the *SetParameters.xml* file that gets generated alongside the deployment package.

XML

```xml
<parameters>
  ...
  <setParameter
    name="ContactService Service Endpoint Address"
    value="http://localhost/ContactManagerService" />
  ...
</parameters>
```

If you install the deployment package manually, IIS Manager will prompt you for the service endpoint address alongside the properties that were parameterized automatically. If you install the deployment package by running the *.deploy.cmd* file, you can edit the *SetParameters.xml* file to provide a value for the service endpoint address together with values for the properties that were parameterized automatically.

For full details on how to create a *parameters.xml* file, see How to: Use Parameters to Configure Deployment Settings When a Package is Installed. The procedure named **To use deployment parameters for Web.config file settings** provides step-by-step instructions.

Modifying the SetParameters.xml File

If you plan to deploy the web application package manually—either by running the *.deploy.cmd* file or by running MSDeploy.exe from the command line—there's nothing to stop you manually editing the *SetParameters.xml* file prior to the deployment. However, if you're working on an enterprise-scale solution, you may need to deploy a web application package as part of a larger, automated build and deployment process. In this scenario, you need the Microsoft Build Engine (MSBuild) to modify the *SetParameters.xml* file for you. You can do this by using the MSBuild **XmlPoke** task.

The Contact Manager sample solution illustrates this process. The code examples that follow have been edited to show only the details that are relevant to this example.

> **Note:** For a broader overview of the project file model in the sample solution, and an introduction to custom project files in general, see Understanding the Project File and Understanding the Build Process.

First, the parameter values of interest are defined as properties in the environment-specific project file (for example, *Env-Dev.proj*).

XML

```xml
<PropertyGroup>
  <ContactManagerIisPath Condition=" '$(ContactManagerIisPath)'=='' ">
    DemoSite/ContactManager
  </ContactManagerIisPath>
  <ContactManagerTargetUrl Condition =" '$(ContactManagerTargetUrl)'=='' ">
    http://localhost:85/ContactManagerService/ContactService.svc
  </ContactManagerTargetUrl>
  <MembershipConnectionString Condition=" '$(MembershipConnectionString)'=='' ">
    Data Source=TESTDB1;Integrated Security=true;Initial Catalog=CMAppServices
  </MembershipConnectionString>
</PropertyGroup>
```

> **Note:** For guidance on how to customize the environment-specific project files for your own server environments, see Configure Deployment Properties for a Target Environment.

Next, the *Publish.proj* file imports these properties. Because each *SetParameters.xml* file is associated with a *.deploy.cmd* file, and we ultimately want the project file to invoke each *.deploy.cmd* file, the project file creates an MSBuild *item* for each *.deploy.cmd* file and defines the properties of interest as *item metadata*.

XML

```xml
<ItemGroup>
  <PublishPackages Include="$(_ContactManagerDest)ContactManager.Mvc.deploy.cmd">
    <ParametersXml>
      $(_ContactManagerDest)ContactManager.Mvc.SetParameters.xml
    </ParametersXml>
    <IisWebAppName>
      $(ContactManagerIisPath)
    </IisWebAppName>
    <MembershipDBConnectionName>
      ApplicationServices-Web.config Connection String
    </MembershipDBConnectionName>
    <MembershipDBConnectionString>
      $(MembershipConnectionString.Replace(";","%3b"))
    </MembershipDBConnectionString>
    <ServiceEndpointParamName>
      ContactService Service Endpoint Address
    </ServiceEndpointParamName>
    <ServiceEndpointValue>
      $(ContactManagerTargetUrl)
    </ServiceEndpointValue>
  </PublishPackages>
  ...
</ItemGroup>
```

In this case:

- The **ParametersXml** metadata value indicates the location of the *SetParameters.xml* file.
- The **IisWebAppName** value is the IIS path to which you want to deploy the web application.

- The **MembershipDBConnectionString** value is the connection string for the membership database, and the **MembershipDBConnectionName** value is the **name** attribute of the corresponding parameter in the *SetParameters.xml* file.

- The **ServiceEndpointValue** value is the endpoint address for the WCF service on the destination server, and the **ServiceEndpointParamName** value is the name attribute of the corresponding parameter in the *SetParameters.xml* file.

Finally, in the *Publish.proj* file, the **PublishWebPackages** target uses the **XmlPoke** task to modify these values in the *SetParameters.xml* file.

XML

```
<Target Name="PublishWebPackages" Outputs="%(PublishPackages.Identity)">
  <XmlPoke
    XmlInputPath="%(PublishPackages.ParametersXml)"
    Query="//parameters/setParameter[@name='%(PublishPackages.ConnectionName)']
           /@value"
    Value="%(PublishPackages.ConnectionString)"
    Condition =" '%(PublishPackages.ConnectionName)'!=''"
  />
  <XmlPoke
    XmlInputPath="%(PublishPackages.ParametersXml)"
    Query="//parameters/setParameter
           [@name='%(PublishPackages.MembershipDBConnectionName)']/@value"
    Value='%(PublishPackages.MembershipDBConnectionString)'
    Condition =" '%(PublishPackages.MembershipDBConnectionName)'!=''"
  />
  <XmlPoke
    XmlInputPath="%(PublishPackages.ParametersXml)"
    Query="//parameters/setParameter[@name='IIS Web Application Name']/@value"
    Value="%(PublishPackages.IisWebAppName)"
    Condition =" '%(PublishPackages.IisWebAppName)'!=''"
  />
  <XmlPoke
    XmlInputPath="%(PublishPackages.ParametersXml)"
    Query="//parameters/setParameter
           [@name='%(PublishPackages.ServiceEndpointParamName)']/@value"
    Value="%(PublishPackages.ServiceEndpointValue)"
    Condition =" '%(PublishPackages.ServiceEndpointParamName)'!=''"
  />
  <!--Execute the .deploy.cmd file-->
  ...
</Target>
```

You'll notice that each **XmlPoke** task specifies four attribute values:

- The **XmlInputPath** attribute tells the task where to find the file you want to modify.

- The **Query** attribute is an XPath query that identifies the XML node you want to change.

- The **Value** attribute is the new value you want to insert into the selected XML node.
- The **Condition** attribute is the criteria on which the task should run or not run. In these cases, the condition ensures that you don't try to insert a null or empty value into the *SetParameters.xml* file.

Conclusion

This topic described the role of the *SetParameters.xm*l file and explained how it's generated when you build a web application project. It explained how you can parameterize additional settings by adding a *parameters.xml* file to your project. It also described how you can modify the *SetParameters.xml* file as part of a larger, automated build process, by using the **XmlPoke** task in your project files.

The next topic, Deploying Web Packages, describes how you can deploy a web package either by running the *.deploy.cmd* file or by using MSDeploy.exe commands directly. In both cases, you can specify your *SetParameters.xml* file as a deployment parameter.

Further Reading

For information on how to create web packages, see Building and Packaging Web Application Projects. For guidance on how to actually deploy a web package, see Deploying Web Packages. For a step-by-step walkthrough on how to create a *parameters.xml* file, see How to: Use Parameters to Configure Deployment Settings When a Package is Installed.

For more general information on parameterization in Web Deploy, see Web Deploy Parameterization in Action (blog post).

Deploying Web Packages

This topic describes how you can publish web deployment packages to a remote server by using the Internet Information Services (IIS) Web Deployment Tool (Web Deploy) 2.0.

There are two main ways in which you can deploy a web package to a remote server:

- You can use the MSDeploy.exe command-line utility directly.
- You can run the *[project name].deploy.cmd* file that the build process generates.

The end result is the same regardless of which approach you use. Essentially, all the *.deploy.cmd* file does is to run MSDeploy.exe with some predetermined values, so that you don't have to provide as much information in order to deploy the package. This simplifies the deployment process. On the other hand, using MSDeploy.exe directly gives you a lot more flexibility over exactly how your package is deployed.

Which approach you use will depend on a variety of factors, including how much control you require over the deployment process and whether you’re targeting the Web Deploy Remote Agent service or the Web Deploy Handler. This topic explains how to use each approach and identifies when each approach is appropriate.

The tasks and walkthroughs in this topic assume that:

- You've built and packaged your web application, as described in Building and Packaging Web Application Projects.
- You've modified the *SetParameters.xml* file to provide the right parameter values for your target environment, as described in Configuring Parameters for Web Package Deployment.

Using the .Deploy.cmd File

Running the [*project name*].*deploy.cmd* file is the simplest way to deploy a web package. In particular, using the *.deploy.cmd* file offers these advantages over using MSDeploy.exe directly:

- You don't need to specify the location of the web deployment package—the *.deploy.cmd* file already knows where it is.
- You don't need to specify the location of the *SetParameters.xml* file—the *.deploy.cmd* file already knows where it is.
- You don't need to specify source and destination MSDeploy providers—the *.deploy.cmd* file already knows which values to use.
- You don't need to specify MSDeploy operation settings—the *.deploy.cmd* file adds the commonly required values to the MSDeploy.exe command automatically.

Before you use the *.deploy.cmd* file to deploy a web package, you should ensure that:

- The *.deploy.cmd* file, the [*project name*].*SetParameters.xml* file, and the web package ([*project name*].*zip*) are in the same folder.
- Web Deploy (MSDeploy.exe) is installed on the computer that runs the *.deploy.cmd* file.

The *.deploy.cmd* file supports various command-line options. When you run the file from a command prompt, this is the basic syntax:

```
[project name].deploy.cmd [/T | /Y]
                          [/M:<computer name>]
                          [/A:<Basic | NTLM>]
                          [/U:<user name>]
                          [/P:<password>]
                          [/L]
                          [/G:<true | false>]
                          [Additional MSDeploy.exe flags]
```

You must specify either a **/T** flag or a **/Y** flag, to indicate whether you want to perform a trial run or a live deployment respectively (don't use both flags in the same command). This table explains the purpose of each of these flags.

Flag	Description
/T	Calls MSDeploy.exe with the **–whatif** flag, which indicates a trial run. Rather than deploying the package, it creates a report of what would happen if you did deploy the package.
/Y	Calls MSDeploy.exe without the **–whatif** flag. This deploys the package to the local computer or the specified destination server.
/M	Specifies the destination server name or service URL. For more information on the values you can provide here, see the **Endpoint Considerations** section in this topic. If you omit the **/M** flag, the package will be deployed to the local computer.
/A	Specifies the authentication type that MSDeploy.exe should use to perform the deployment. Possible values are **NTLM** and **Basic**. If you omit the **/A** flag, the authentication type defaults to **NTLM** for deployment to the Web Deploy Remote Agent service and to **Basic** for deployment to the Web Deploy Handler.
/U	Specifies the user name. This applies only if you're using basic authentication.
/P	Specifies the password. This applies only if you're using basic authentication.
/L	Indicates that the package should be deployed to the local IIS Express instance.
/G	Specifies that the package is deployed using the tempAgent provider setting. If you omit the **/G** flag, the value defaults to **false**.

Note: Every time the build process creates a web package, it also creates a file named *[project name].deploy-readme.txt* that explains these deployment options.

In addition to these flags, you can specify Web Deploy operation settings as additional *.deploy.cmd* parameters. Any additional settings you specify are simply passed through to the underlying MSDeploy.exe command. For more information on these settings, see Web Deploy Operation Settings.

Suppose you want to deploy the ContactManager.Mvc web application project to a test environment by running the *.deploy.cmd* file. Your test environment is configured to use the Web Deploy Remote Agent service, as described in Configure a Web Server for Web Deploy Publishing (Remote Agent). To deploy the web application, you need to complete the next steps.

To deploy a web application using the .deploy.cmd file

1. Build and package the web application project, as described in Building and Packaging Web Application Projects.
2. Modify the *ContactManager.Mvc.SetParameters.xml* file to contain the correct parameter values for your test environment, as described in Configuring Parameters for Web Package Deployment.

3. Open a Command Prompt window and navigate to the location of the *ContactManager.Mvc.deploy.cmd* file.

4. Type this command, and then press Enter:

```
ContactManager.Mvc.deploy.cmd /Y /M:TESTWEB1 /A:NTLM
```

In this example:

- The **/Y** flag indicates that you want to actually deploy the package, rather than doing a trial run.
- The **/M** flag indicates that you want to deploy the package to the server named TESTWEB1. From this value, MSDeploy.exe will attempt to deploy the package to the Web Deploy Remote Agent service at http://TESTWEB1/MSDeployAgentService.
- The **/A** flag indicates that you want to use NTLM authentication. As such, you don't need to specify a user name and password.

To illustrate how using the *.deploy.cmd* file simplifies the deployment process, take a look at the MSDeploy.exe command that gets generated and executed when you run *ContactManager.Mvc.deploy.cmd* using the options shown above.

```
msdeploy.exe
-source:package='C:\Users\matt.FABRIKAM\Desktop\ContactManager-03\ContactManager\
 Publish\Out\_PublishedWebsites\ContactManager.Mvc_Package\ContactManager.Mvc.zip' -
dest:auto,computerName='TESTWEB1.fabrikam.net', authtype='NTLM',
 includeAcls='False'
-verb:sync
-disableLink:AppPoolExtension
-disableLink:ContentExtension
-disableLink:CertificateExtension
-setParamFile:"C:\Users\matt.FABRIKAM\Desktop\ContactManager-03\ContactManager\
 Publish\Out\_PublishedWebsites\ContactManager.Mvc_Package\
 ContactManager.Mvc.SetParameters.xml"
```

For more information on using the *.deploy.cmd* file to deploy a web package, see How to: Install a Deployment Package Using the deploy.cmd File.

Using MSDeploy.exe

Although using the *.deploy.cmd* file generally simplifies the deployment process, there are some situations when it's preferable to use MSDeploy.exe directly. For example:

- If you want to deploy to the Web Deploy Handler as a non-administrator user, you can’t use the *.deploy.cmd* file. This is due to a bug in Web Deploy 2.0, as described under **Endpoint Considerations**.
- If you want to manually switch between different *SetParameters.xml* files in different locations, you may prefer to use MSDeploy.exe directly.

- If you want to override several MSDeploy.exe command-line arguments, you may prefer to use MSDeploy.exe directly.

When you use MSDeploy.exe, you need to provide three key pieces of information:

- A **–source** parameter that indicates where your data is coming from.
- A **–dest** parameter that indicates where your data is going to.
- A **–verb** parameter that indicates the operation you want to perform.

MSDeploy.exe relies on Web Deploy providers to process source and destination data. Web Deploy includes a lot of providers that represent the range of applications and data sources it can work with—for example, there are providers for SQL Server databases, IIS web servers, certificates, global assembly cache (GAC) assemblies, various different configuration files, and lots of other types of data. Both the **–source** parameter and the **–dest** parameter must specify a provider, in the form **–source**:[*providerName*]=[*location*]. When you're deploying a web package to an IIS website, you should use these values:

- The **–source** provider is always package. For example:

  ```
  -source:package='[path to web package]'
  ```

- The **–dest** provider is always auto. For example:

  ```
  -dest:auto='[server name or service URL]'
  ```

- The **–verb** is always **sync**.

  ```
  -verb:sync
  ```

In addition, you'll need to specify various other provider-specific settings and general operation settings. For example, suppose you want to deploy the ContactManager.Mvc web application to a staging environment. The deployment will target the Web Deploy Handler and must use basic authentication. To deploy the web application, you need to complete the next steps.

To deploy a web application using MSDeploy.exe

1. Build and package the web application project, as described in Building and Packaging Web Application Projects.
2. Modify the *ContactManager.Mvc.SetParameters.xml* file to contain the correct parameter values for your staging environment, as described in Configuring Parameters for Web Package Deployment.
3. Open a Command Prompt window and browse to the location of MSDeploy.exe. This is typically at %PROGRAMFILES%\IIS\Microsoft Web Deploy V2\msdeploy.exe.
4. Type this command, and then press Enter (disregard the line breaks):

```
MSDeploy.exe
  -source:package="[path]\ContactManager.Mvc.zip"
  -dest:auto,
        computerName="https://stageweb1:8172/MSDeploy.axd?site=DemoSite",
        username="FABRIKAM\stagingdeployer",
        password="Pa$$w0rd",
        authtype="Basic",
        includeAcls="False"
  -verb:sync
  -disableLink:AppPoolExtension
  -disableLink:ContentExtension
  -disableLink:CertificateExtension
  -setParamFile:"[path]\ContactManager.Mvc.SetParameters.xml"
  -allowUntrusted
```

In this example:

- The **–source** parameter specifies the **package** provider and indicates the location of the web package.
- The **–dest** parameter specifies the **auto** provider. The **computerName** setting provides the service URL of the Web Deploy Handler on the destination server. The **authtype** setting indicates that you want to use basic authentication, and as such you need to provide a **username** and a **password**. Finally, the **includeAcls="False"** setting indicates that you don't want to copy the access control lists (ACLs) of the files in your source web application to the destination server.
- The **–verb:sync** argument indicates that you want to replicate the source content on the destination server.
- The **–disableLink** arguments indicate that you don't want to replicate application pools, virtual directory configuration, or Secure Sockets Layer (SSL) certificates on the destination server. For more information, see Web Deploy Link Extensions.
- The **–setParamFile** parameter provides the location of the *SetParameters.xml* file.
- The **–allowUntrusted** switch indicates that Web Deploy should accept SSL certificates that were not issued by a trusted certification authority. If you're deploying to the Web Deploy Handler, and you've used a self-signed certificate to secure the service URL, you need to include this switch.

Automating Web Package Deployment

In a lot of enterprise scenarios, you'll want to deploy your web packages as part of a larger single-step or automated deployment. Regardless of whether you choose to deploy your web packages by running the *.deploy.cmd* file or by using MSDeploy.exe directly, you can parameterize your commands and call them from a target in a Microsoft Build Engine (MSBuild) project file.

In the Contact Manager sample solution, take a look at the **PublishWebPackages** target in the *Publish.proj* file. This target runs once for each *.deploy.cmd* file identified by an item list named **PublishPackages**. The target uses properties and item metadata to build up a full set of argument values for each *.deploy.cmd* file and then uses the **Exec** task to run the command.

XML

```xml
<Target Name="PublishWebPackages" Outputs="%(PublishPackages.Identity)">
  ...
  <PropertyGroup>
    <_WhatIfSwitch>/Y</_WhatIfSwitch>
    <_WhatIfSwitch Condition=" '$(_WhatIf)'=='true' ">/T</_WhatIfSwitch>
    <_Cmd>
      %(PublishPackages.FullPath) $(_WhatifSwitch) /M:$(MSDeployComputerName)
      /U:$(MSDeployUsername) /P:$(Password) /A:$(MSDeployAuth)
      %(PublishPackages.AdditionalMSDeployParameters)
    </_Cmd>
  </PropertyGroup>
  <Exec Command="$(_Cmd)"/>
</Target>
```

> **Note:** For a broader overview of the project file model in the sample solution, and an introduction to custom project files in general, see Understanding the Project File and Understanding the Build Process.

Endpoint Considerations

Regardless of whether you deploy your web package by running the *.deploy.cmd* file or by using MSDeploy.exe directly, you need to specify a computer name or a service endpoint for your deployment.

If the destination web server is configured for deployment using the Web Deploy Remote Agent service, you specify the target service URL as your destination.

```
http://[server name]/MSDeployAgentService
```

Alternatively, you can specify the server name alone as your destination, and Web Deploy will infer the remote agent service URL.

```
[server name]
```

If the destination web server is configured for deployment using the Web Deploy Handler, you need to specify the endpoint address of the IIS Web Management Service (WMSvc) as your destination. By default, this takes the form:

```
https://[server name]:8172/MSDeploy.axd
```

You can target any of these endpoints using either the *.deploy.cmd* file or MSDeploy.exe directly. However, if you want to deploy to the Web Deploy Handler as a non-administrator user, as described in

Configure a Web Server for Web Deploy Publishing (Web Deploy Handler), you need to add a query string to the service endpoint address.

```
https://[server name]:8172/MSDeploy.axd?site=[IIS website name]
```

This is because the non-administrator user doesn't have server-level access to IIS; he or she only has access to a specific IIS website. At the time of writing, due to a bug in the Web Publishing Pipeline (WPP), you can't run the *.deploy.cmd* file using an endpoint address that includes a query string. In this scenario, you need to deploy your web package by using MSDeploy.exe directly.

> **Note:** For more information on the Web Deploy Remote Agent service and the Web Deploy Handler, see Choosing the Right Approach to Web Deployment. For guidance on how to configure your environment-specific project files to deploy to these endpoints, see Configure Deployment Properties for a Target Environment.

Authentication Considerations

Regardless of whether you deploy your web package by running the *.deploy.cmd* file or by using MSDeploy.exe directly, you need to specify an authentication type. Web Deploy accepts two possible values: **NTLM** or **Basic**. If you specify basic authentication, you also need to provide a user name and password. There are various factors you need to be aware of when you select an authentication type:

- If you're deploying to the Web Deploy Remote Agent service, you must use NTLM authentication. The remote agent service doesn't accept basic authentication credentials.
- If you're deploying to the Web Deploy Handler, you can use either NTLM or basic authentication. The default setting is basic authentication. Although basic authentication relies on user names and passwords being transmitted in plain text, your credentials are protected as the Web Deploy Handler always uses SSL encryption.
- If your web package includes a database, and the web server and database server are separate machines, you won’t be able to deploy the database using NTLM authentication due to the NTLM "double-hop" limitation. You need to either use SQL Server credentials in your deployment connection string or supply basic authentication credentials to Web Deploy. This issue is described in more detail in Deploying Membership Databases to Enterprise Environments.

Conclusion

This topic described how you can deploy a web package either by running the *.deploy.cmd* file or by using MSDeploy.exe directly. It explained when each approach might be appropriate, and it described how you can parameterize and run a deployment command as part of a larger single-step or automated build process.

Further Reading

For guidance on how to create and parameterize a web deployment package, see Building and Packaging Web Application Projects and Configuring Parameters for Web Package Deployment. For guidance on how to build and deploy web packages from a Team Foundation Server (TFS) instance, see Configuring Team Foundation Server for Automated Web Deployment. For information on how to customize and troubleshoot the deployment process, see Excluding Files and Folders from Deployment.

Deploying Database Projects

This topic explains the build and deployment process for database projects in Visual Studio 2010. It also describes how you can use the Microsoft Build Engine (MSBuild) and VSDBCMD.exe to gain more control over database deployment.

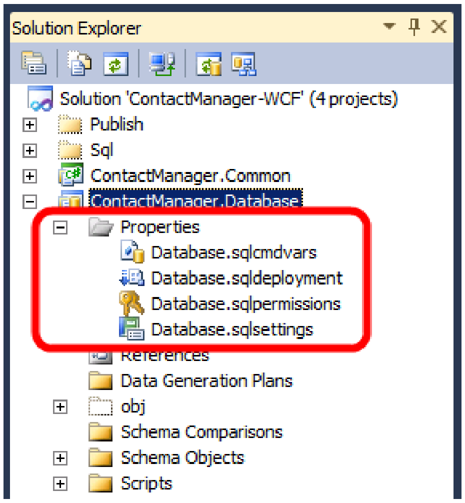
If you want to control how your database projects are deployed, and customize your deployment for different destination environments, you first need to understand how the build process works and what deployment options are available to you. This topic will help you to understand these key aspects of building and deploying database projects:

- What are the inputs to the build process?
- What are the outputs from the build process?
- What are the deployment options for database projects?
- What are the touch points for modifying a database project deployment?

Note: In lots of enterprise deployment scenarios, you need the ability to publish incremental updates to a deployed database. The alternative is to recreate the database on every deployment, which means you lose any data in the existing database. When you work with Visual Studio 2010, using VSDBCMD is the recommended approach to incremental database publishing. However, the next version of Visual Studio and the Web Publishing Pipeline (WPP) will include tooling that supports incremental publishing directly.

Understanding the Build Process

If you open the Contact Manager sample solution in Visual Studio 2010, you'll see that the database project includes a Properties folder that contains four files.



Together with the project file (*ContactManager.Database.dbproj* in this case), these files control various aspects of the build and deployment process:

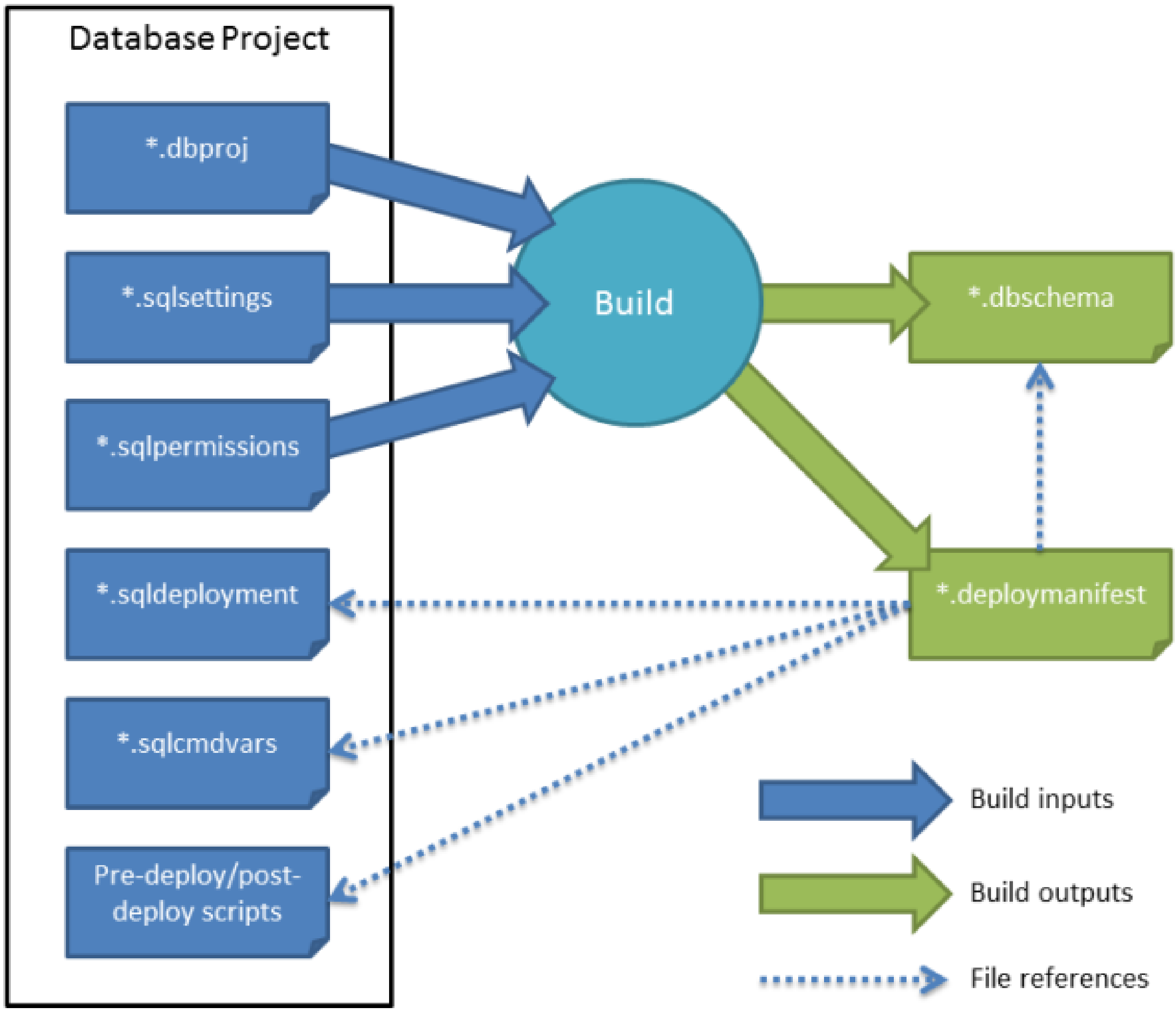
- The *Database.sqlcmdvars* file provides values for any SQLCMD variables you use when you deploy the project. Each solution configuration (for example, debug and release) can specify a different .sqlcmdvars file.
- The *Database.sqldeployment* file provides deployment-specific settings, like whether to use the collation defined in your project or the collation of the destination server, whether to recreate the destination database every time or simply amend the existing database to bring it up to date, and so on. Each solution configuration can specify a different .sqldeployment file.
- The *Database.sqlpermissions* file is an XML document that you can use to define any permissions you want to add to the target database. All solution configurations share the same .sqlpermissions file.
- The *Database.sqlsettings* file specifies the database-level properties to use when creating the database, like the collation to use, the behavior of comparison operators, and so on. All solution configurations share the same .sqlsettings file.

It's worth taking a moment to open these files in Visual Studio and familiarize yourself with the contents.

When you build a database project, the build process creates two files:

- A *database schema* (.dbschema file). This describes the schema of the database you want to create in XML format.
- A *deployment manifest* (.deploymanifest file). This contains all the information required to create and deploy your database. It references the .dbschema file along with other resources, like the deployment instructions (the .sqldeployment file) and any pre-deployment or post-deployment SQL scripts.

This shows the relationship between these resources:



As you can see, the .sqlsettings file and the .sqlpermissions file are inputs to the build process. Along with the database project file, these files are used to create the database schema file. The .sqldeployment file and the .sqlcmdvars file pass through the build process unchanged. The deployment manifest indicates the location of the database schema, the .sqldeployment file, the .sqlcmdvars file, and any pre-deployment or post-deployment SQL scripts.

Why Use VSDBCMD to Deploy a Database Project?

There are various different approaches to deploying database projects. However, not all of them are suitable for deploying a database project to remote servers in an enterprise environment. Consider what you want from a database project deployment. In enterprise deployment scenarios, you're likely to want:

- The ability to deploy the database project from a remote location.

- The ability to make incremental updates to an existing database.
- The ability to include pre-deployment scripts or post-deployment scripts.
- The ability to tailor the deployment to multiple destination environments.
- The ability to deploy the database project as part of a larger, typically scripted, single-step solution deployment.

There are three main approaches you can use to deploy a database project:

- You can use the deployment functionality with the database project type in Visual Studio 2010. When you build and deploy a database project in Visual Studio 2010, the deployment process uses the deployment manifest to generate a SQL-based deployment file specific to the build configuration. This will create the database if it doesn't already exist or make any necessary changes to the database if it does already exist. You can use SQLCMD.exe to run this file on your destination server, or you can set Visual Studio to create and run the file. The disadvantage of this approach is that you have only limited control over the deployment settings. You may often also need to modify the SQL deployment file to provide environment-specific variable values. You can only use this approach from a computer with Visual Studio 2010 installed, and the developer would need to know and provide connection strings and credentials for all destination environments.
- You can use the Internet Information Services (IIS) Web Deployment Tool (Web Deploy) to deploy a database as part of a web application project. However, this approach is a lot more complex if you want to deploy a database project rather than simply replicate an existing local database on a destination server. You can configure Web Deploy to run the SQL deployment script that the database project generates, but in order to do this, you need to create a custom WPP targets file for your web application project. This adds a substantial amount of complexity to the deployment process. In addition, Web Deploy does not directly support incremental updates to existing databases. For more information on this approach, see Extending the Web Publishing Pipeline to package database project deployed SQL file.
- You can use the VSDBCMD utility to deploy the database, using either the database schema or the deployment manifest. You can call VSDBCMD.exe from an MSBuild target, which lets you publish databases as part of a larger, scripted deployment process. You can override the variables in your .sqlcmdvars file and lots of other database properties from a VSDBCMD command, which allows you to customize your deployment for different environments without creating multiple build configurations. VSDBCMD provides differentiation functionality, which means it will make only the necessary changes to align a destination database with your database schema. VSDBCMD also offers a wide range of command-line options, which give you fine-grained control over the deployment process.

From this overview, you can see that using VSDBCMD with MSBuild is the approach best suited to a typical enterprise deployment scenario:

	Visual Studio 2010	Web Deploy 2.0	VSDBCMD.exe
Supports remote deployment?	Yes	Yes	Yes
Supports incremental updates?	Yes	No	Yes
Supports pre/post-deployment scripts?	Yes	Yes	Yes
Supports multi-environment deployment?	Limited	Limited	Yes
Supports scripted deployment?	Limited	Yes	Yes

The remainder of this topic describes the use of VSDBCMD with MSBuild to deploy database projects.

Understanding the Deployment Process

The VSDBCMD utility lets you deploy a database using either the database schema (the .dbschema file) or the deployment manifest (the .deploymanifest file). In practice, you'll almost always use the deployment manifest, as the deployment manifest lets you provide default values for various deployment properties and identify any pre-deployment or post-deployment SQL scripts you want to run. For example, this VSDBCMD command is used to deploy the **ContactManager** database to a database server in a test environment:

```
vsdbcmd.exe /a:Deploy
            /manifest:"…\ContactManager.Database.deploymanifest"
            /cs:"Data Source=TESTDB1;Integrated Security=true"
            /p:TargetDatabase=ContactManager
            /dd+
            /script:"…\Publish-ContactManager-Db.sql"
```

In this case:

- The **/a** (or **/Action**) switch specifies what you want VSDBCMD to do. You can set this to **Import** or **Deploy**. The **Import** option is used to generate a .dbschema file from an existing database, and the **Deploy** option is used to deploy a .dbschema file to a target database.
- The **/manifest** (or **/ManifestFile**) switch identifies the .deploymanifest file you want to deploy. If you wanted to use the .dbschema file instead, you'd use the **/model** (or **/ModelFile**) switch.
- The **/cs** (or **/ConnectionString**) switch provides the connection string for the target database server. Note that this doesn't include the name of the database—VSDBCMD needs to connect to the server to create the database; it doesn't need to connect to an individual database. If your .deploymanifest file includes a connection string, you can omit this switch. If you use the switch anyway, the switch value will override the .deploymanifest value.
- The **/p:TargetDatabase** property provides the name you want to assign to the target database on creation. This overrides the value of the **TargetDatabase** property in the .deploymanifest file.

You can use the **/p:***[property name]* syntax to set a wide variety of deployment properties and to override any SQLCMD variables declared in your .sqlcmdvars file.

- The **/dd+** (or **/DeployToDatabase+**) switch indicates that you want to create a deployment and deploy it to the target environment. If you specify **/dd-**, or omit the switch, VSDBCMD will generate a deployment script but will not deploy it to the target environment. This switch is often the source of confusion and is explained in more detail in the next section.
- The **/script** (or **/DeploymentScriptFile**) switch specifies where you want to generate the deployment script. This value does not affect the deployment process.

For more information on VSDBCMD, see Command-Line Reference for VSDBCMD.EXE (Deployment and Schema Import) and How to: Prepare a Database for Deployment From a Command Prompt by Using VSDBCMD.EXE.

For an example of how you can use VSDBCMD from an MSBuild project file, see Understanding the Build Process. For examples of how to configure database deployment settings for multiple environments, see Customizing Database Deployments for Multiple Environments.

Understanding the DeployToDatabase Switch

The behavior of the **/dd** or **/DeployToDatabase** switch depends on whether you're using VSDBCMD with a .dbschema file or a .deploymanifest file. If you're using a .dbschema file, the behavior is fairly straightforward:

- If you specify **/dd+** or **/dd**, VSDBCMD will generate a deployment script and deploy the database.
- If you specify **/dd-** or omit the switch, VSDBCMD will generate a deployment script only.

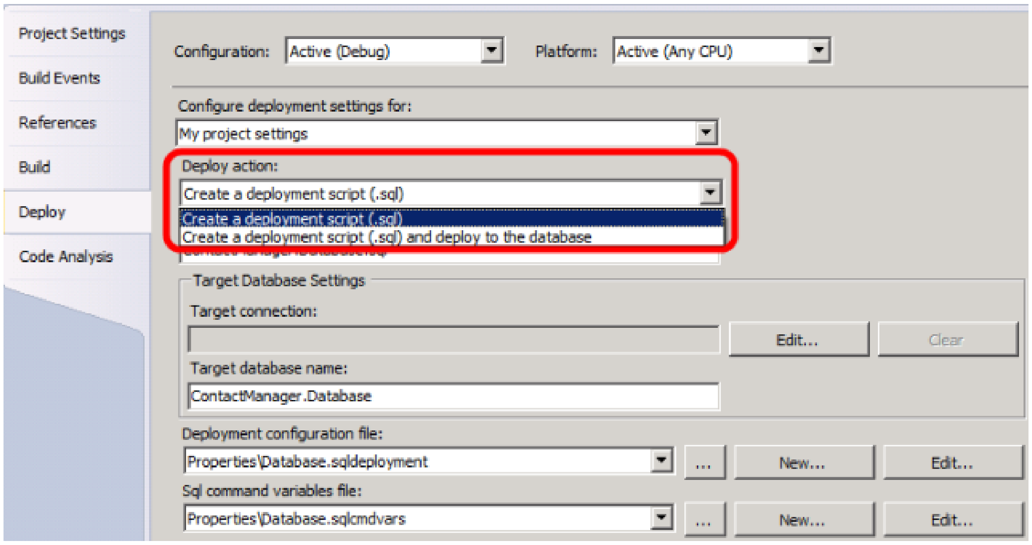
If you're using a .deploymanifest file, the behavior is a lot more complicated. This is because the .deploymanifest file contains a property name **DeployToDatabase** that also determines whether the database is deployed.

XML

```
<DeployToDatabase>False</DeployToDatabase>
```

The value of this property is set according to the properties of the database project. If you set the **Deploy action** to **Create a deployment script (.sql)**, the value will be **False**. If you set the **Deploy action** to **Create a deployment script (.sql) and deploy to the database**, the value will be **True**.

> **Note:** These settings are associated with a specific build configuration and platform. For example, if you configure settings for the **Debug** configuration and then publish using the **Release** configuration, your settings will not be used.



Configuration: Active (Debug)

Platform: Active (Any CPU)

Configure deployment settings for: My project settings

Deploy action: Create a deployment script (.sql)

Create a deployment script (.sql)

Create a deployment script (.sql) and deploy to the database

Target Database Settings

Target connection:

Target database name: ContactManager.Database

Deployment configuration file: Properties\Database.sqldeployment

Sql command variables file: Properties\Database.sqlcmdvars

Note: In this scenario, the **Deploy action** should always be set to **Create a deployment script (.sql)**, because you don't want Visual Studio 2010 to deploy your database. In other words, the **DeployToDatabase** property should always be **False**.

When a **DeployToDatabase** property is specified, the **/dd** switch will only override the property if the property value is **false**:

- If the **DeployToDatabase** property is **False**, and you specify **/dd+** or **/dd**, VSDBCMD will override the **DeployToDatabase** property and deploy the database.
- If the **DeployToDatabase** property is **False**, and you specify **/dd-** or omit the switch, VSDBCMD will not deploy the database.
- If the **DeployToDatabase** property is **True**, VSDBCMD will ignore the switch and deploy the database.
- A deployment script is generated in each case, regardless of whether you're deploying the database as well.

Conclusion

This topic provided an overview of the build and deployment process for database projects in Visual Studio 2010. It also described how you can use VSDBCMD.exe with MSBuild to support enterprise-scale database deployment.

For more information on how this works in practice, see Customizing Database Deployments for Multiple Environments.

Further Reading

For information on how to customize database deployments by creating a separate deployment configuration file for each environment, see Customizing Database Deployments for Multiple

Environments. For guidance on how to configure database role memberships by running a post-deployment script, see Deploying Database Role Memberships to Test Environments. For guidance on managing some of the unique challenges that membership databases impose, see Deploying Membership Databases to Enterprise Environments.

These topics on MSDN provide broader guidance and background information on Visual Studio database projects and the database deployment process:

- Visual Studio 2010 SQL Server Database Projects
- Managing Database Change
- How to: Prepare a Database for Deployment From a Command Prompt by Using VSDBCMD.EXE
- An Overview of Database Build and Deployment

Creating and Running a Deployment Command File

This topic describes how to build a command file that will let you run a deployment using Microsoft Build Engine (MSBuild) project files as a single-step, repeatable process.

Process Overview

In this topic, you'll learn how to create and run a command file that uses these project files to perform a repeatable deployment to your target environment. Essentially, the command file simply needs to contain an MSBuild command that:

- Tells MSBuild to execute the environment-agnostic *Publish.proj* file.
- Tells the *Publish.proj* file which file contains the environment-specific project settings and where to find it.

Create an MSBuild Command

As described in Understanding the Build Process, the environment-specific project file—for example, *Env-Dev.proj*—is designed to be imported into the environment-agnostic *Publish.proj* file at build time. Together, these two files provide a complete set of instructions that tell MSBuild how to build and deploy your solution.

The *Publish.proj* file uses an **Import** element to import the environment-specific project file.

XML

```xml
<Import Project="$(TargetEnvPropsFile)"/>
```

As such, when you use MSBuild.exe to build and deploy the Contact Manager solution, you need to:

- Run MSBuild.exe on the *Publish.proj* file.

- Specify the location of the environment-specific project file by supplying a command-line parameter named **TargetEnvPropsFile**.

To do this, your MSBuild command should resemble this:

```
msbuild.exe Publish.proj /p:TargetEnvPropsFile=EnvConfig\Env-Dev.proj
```

From here, it's a simple step to move to a repeatable, single-step deployment. All you need to do is to add your MSBuild command to a .cmd file. In the Contact Manager solution, the Publish folder includes a file named *Publish-Dev.cmd* that does exactly this.

```
%windir%\Microsoft.NET\Framework\v4.0.30319\msbuild.exe Publish.proj /fl
/p:TargetEnvPropsFile=EnvConfig\Env-Dev.proj
echo
pause
```

Note: The **/fl** switch instructs MSBuild to create a log file named *msbuild.log* in the working directory in which MSBuild.exe was invoked.

To deploy or redeploy the Contact Manager solution, all you need to do is run the *Publish-Dev.cmd* file. When you run the file, MSBuild will:

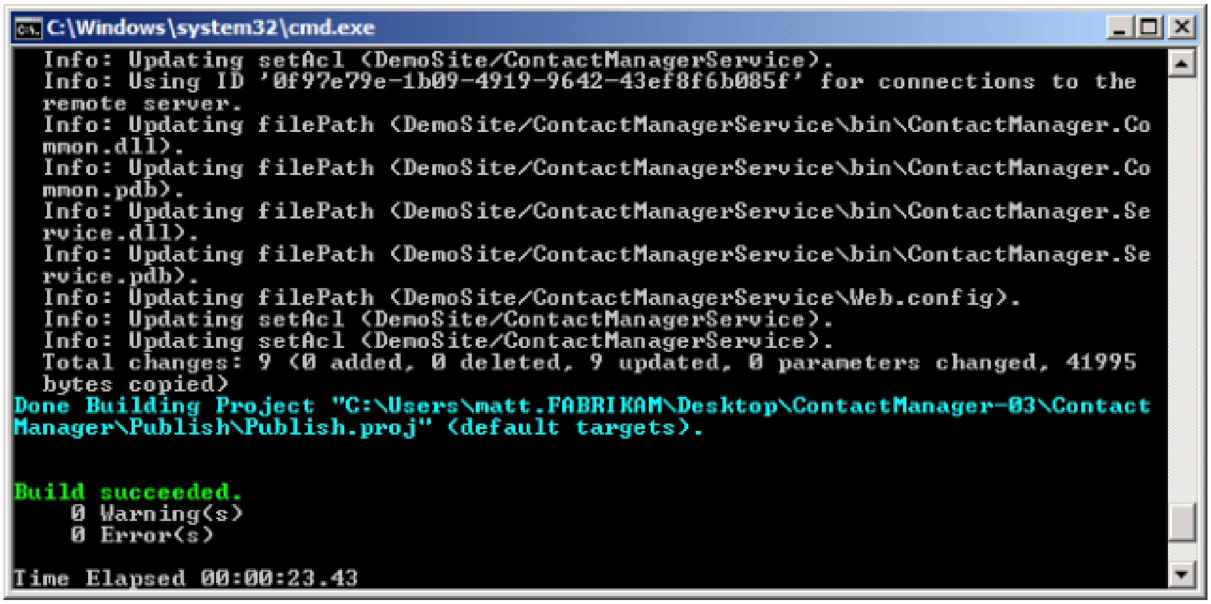
- Build all the projects in the solution.
- Generate deployable web packages for the web application projects.
- Generate .dbschema and .deploymanifest files for the database projects.
- Deploy the web packages to the web server.
- Deploy the database to the database server.

Run the Deployment

When you've created a command file for your target environment, you should be able to complete the entire deployment by simply running the file.

To deploy the Contact Manager solution to your test environment

1. On your developer workstation, open Windows Explorer, and then browse to the location of the *Publish-Dev.cmd* file.
2. Double-click the file to run it.
3. If an **Open File – Security Warning** dialog box appears, click **Run**.
4. If your configuration settings and test servers are set up correctly, the Command Prompt window will show a **Build succeeded** message when MSBuild has finished processing the project files.



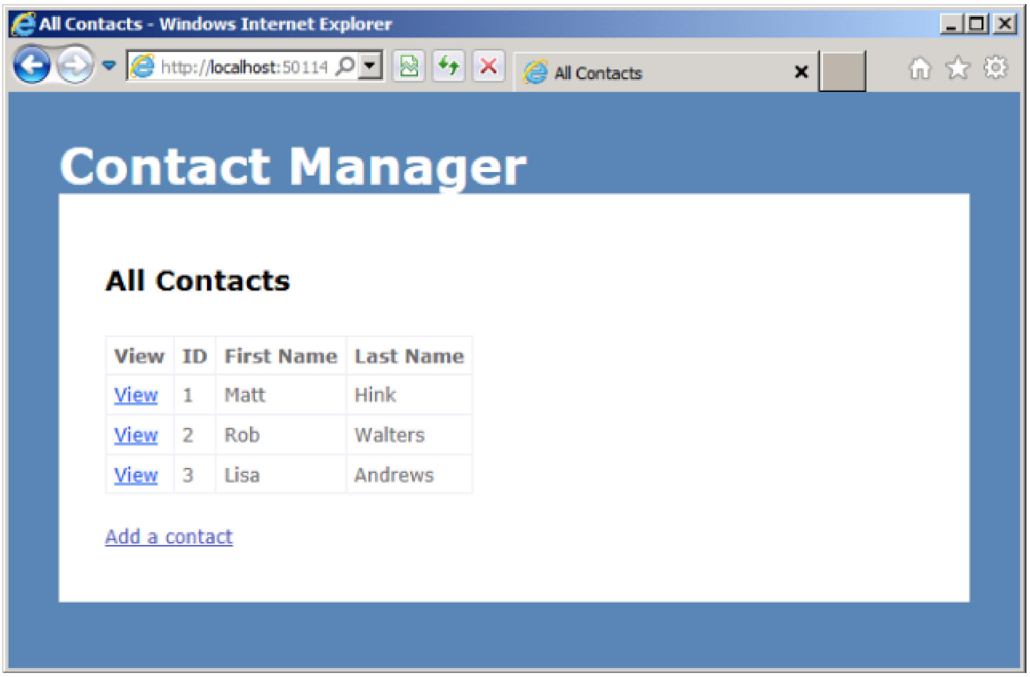
```
  Info: Updating setAcl (DemoSite/ContactManagerService).
  Info: Using ID '0f97e79e-1b09-4919-9642-43ef8f6b085f' for connections to the
  remote server.
  Info: Updating filePath (DemoSite/ContactManagerService\bin\ContactManager.Co
  mmon.dll).
  Info: Updating filePath (DemoSite/ContactManagerService\bin\ContactManager.Co
  mmon.pdb).
  Info: Updating filePath (DemoSite/ContactManagerService\bin\ContactManager.Se
  rvice.dll).
  Info: Updating filePath (DemoSite/ContactManagerService\bin\ContactManager.Se
  rvice.pdb).
  Info: Updating filePath (DemoSite/ContactManagerService\Web.config).
  Info: Updating setAcl (DemoSite/ContactManagerService).
  Info: Updating setAcl (DemoSite/ContactManagerService).
  Total changes: 9 (0 added, 0 deleted, 9 updated, 0 parameters changed, 41995
  bytes copied)
Done Building Project "C:\Users\matt.FABRIKAM\Desktop\ContactManager-03\Contact
Manager\Publish\Publish.proj" (default targets).


Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:23.43
```

5. If this is the first time you've deployed the solution to this environment, you'll need to add the test web server machine account to the **db_datawriter** and **db_datareader** roles on the **ContactManager** database. This procedure is described in Configure a Database Server for Web Deploy Publishing.

> **Note:** You only need to assign these permissions when you create the database. By default, the build process will not recreate the database on every deployment—instead, it will compare the existing database to the latest schema and make only the changes required. As a result, you should only need to map these database roles the first time you deploy the solution.

6. Open Internet Explorer and browse to the URL of the Contact Manager application (for example, **http://testweb1:85/ContactManager/**).

7. Verify that the application works as expected and you're able to add contacts.



Contact Manager

All Contacts

View	ID	First Name	Last Name
View	1	Matt	Hink
View	2	Rob	Walters
View	3	Lisa	Andrews

Add a contact

Conclusion

Creating a command file containing your MSBuild instructions provides you with a quick and easy way of building and deploying a multi-project solution to a specific destination environment. If you need to repeatedly deploy your solution to multiple destination environments, you can create multiple command files. In each command file, the MSBuild command will build the same universal project file, but it will specify a different environment-specific project file. For example, a command file to publish to a developer or test environment might contain this MSBuild command:

```
msbuild.exe Publish.proj /p:TargetEnvPropsFile=EnvConfig\Env-Dev.proj
```

A command file to publish to a staging environment might contain this MSBuild command:

```
msbuild.exe Publish.proj /p:TargetEnvPropsFile=EnvConfig\Env-Stage.proj
```

Note: For guidance on how to customize the environment-specific project files for your own server environments, see Configure Deployment Properties for a Target Environment.

You can also customize the build process for each environment by overriding properties or setting various other switches in your MSBuild command. For more information, see MSBuild Command Line Reference.

Manually Installing Web Packages

This topic describes how to manually import a web deployment package into Internet Information Services (IIS).

The topic Building and Packaging Web Application Projects described how the IIS Web Deployment Tool (Web Deploy), in conjunction with the Microsoft Build Engine (MSBuild) and the Web Publishing Pipeline (WPP), lets you package your web application projects into a single zip file. This file, commonly known as a web deployment package (or simply a deployment package), contains all the content and configuration information that IIS needs in order to re-create your web application on a web server.

Once you've created a web deployment package, you can publish it to an IIS server in various ways. In a lot of scenarios, you'll want to take advantage of the integration points between MSBuild, the WPP, and Web Deploy to create and install web packages remotely as part of an automated or single-step build and deployment process. This process is described in Deploying Web Packages. However, this isn't always possible. Suppose you want to deploy a web application to an Internet-facing production environment. For security reasons, such a production environment is at the very least likely to be behind a firewall on a subnet that is separate from the build server, in a perimeter network (also known as DMZ, demilitarized zone, and screened subnet). In lots of cases, the production environment will be on a separate domain or on a physically isolated network.

In these scenarios, your only option may be to port the web package onto the destination server and manually import it into IIS. Although this approach precludes automated deployment, it's still a highly effective technique for publishing a web application—you simply copy a single zip file to your web server and use a wizard to guide you through the import process.

Task Overview

You'll need to complete these high-level tasks to import a web deployment package into IIS:

- Create a web deployment package using the MSBuild command line, Team Build, or Visual Studio 2010.
- Copy the web package to the destination web server.
- Use the Import Application Package Wizard in IIS Manager to install the web package and provide values for variables like connection strings and service endpoints.

This topic will show you how to perform these procedures. The tasks and walkthroughs in this topic assume that you're already familiar with the concepts behind web packages, Web Deploy, and the WPP. For more information, see Building and Packaging Web Application Projects.

Note: This topic is best used in conjunction with Configure a Web Server for Web Deploy Publishing (Offline Deployment), which explains how to install the required components and prepare an IIS website for package import.

Create a Web Deployment Package

The first task is to create a web deployment package for the web application project you want to deploy. You can create web packages in a variety of ways.

Approach 1: Create a package as part of the build process with Visual Studio

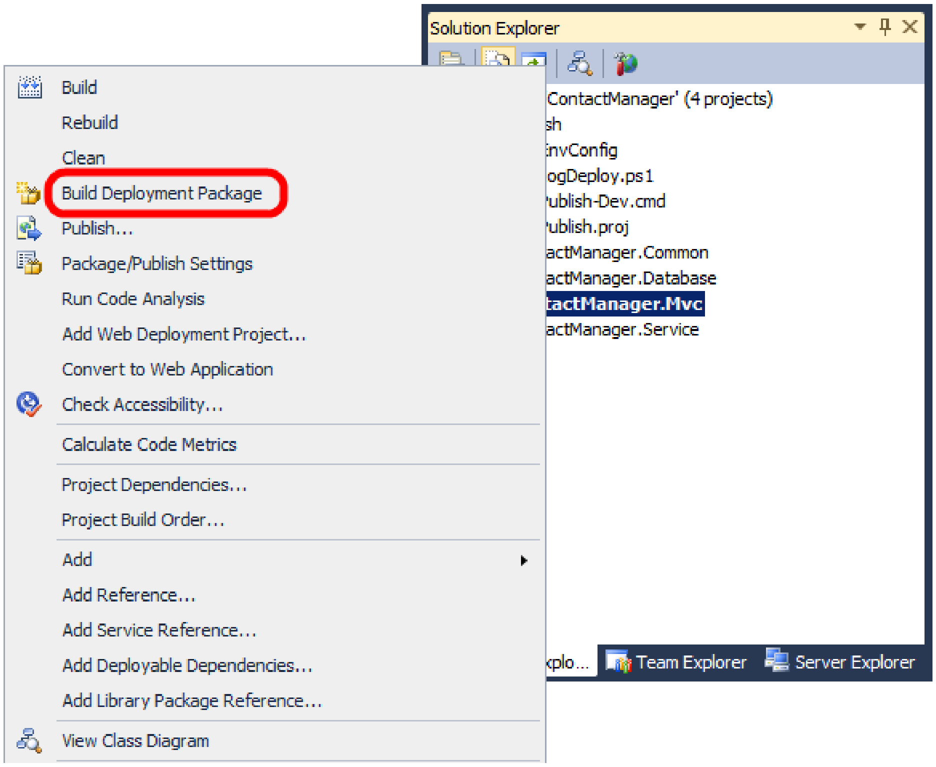
You can configure your web application project to create a web deployment package after every build through the **Package/Publish Web** tab on the project property pages. This process is described in Building and Packaging Web Application Projects.

Approach 2: Create a package as part of the build process with MSBuild

If you build your web application project by using MSBuild directly, either through a custom MSBuild project file or from the command line, you can create a web deployment package as part of the build process by including the **DeployOnBuild=true** and **DeployTarget=Package** properties in your command. This process is described in Understanding the Build Process.

Approach 3: Create a package on demand in Visual Studio

You can create a web deployment package for a web application project at any time in Visual Studio 2010. To do this, in the **Solution Explorer** window, right-click your web application project, and then click **Build Deployment Package**.

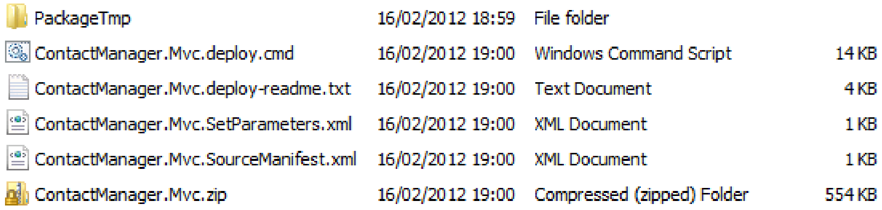


Approach 4: Create a package on demand from the command line

You can create a web deployment package from the command line by invoking the **Package** target on your web application project using MSBuild. The command should resemble this:

```
MSBuild.exe [Path to your project].[csproj/vbproj] /T:Package
```

Whichever approach you use, the end result is the same. The WPP creates a web deployment package as a zip file, together with various supporting resources, in the output folder for your web application project.



PackageTmp	16/02/2012 18:59	File folder	
ContactManager.Mvc.deploy.cmd	16/02/2012 19:00	Windows Command Script	14 KB
ContactManager.Mvc.deploy-readme.txt	16/02/2012 19:00	Text Document	4 KB
ContactManager.Mvc.SetParameters.xml	16/02/2012 19:00	XML Document	1 KB
ContactManager.Mvc.SourceManifest.xml	16/02/2012 19:00	XML Document	1 KB
ContactManager.Mvc.zip	16/02/2012 19:00	Compressed (zipped) Folder	554 KB

When you're planning to import the web package manually, you require only the zip file. Copy this file to your target web server and you can begin the import process.

Import a Web Package into IIS

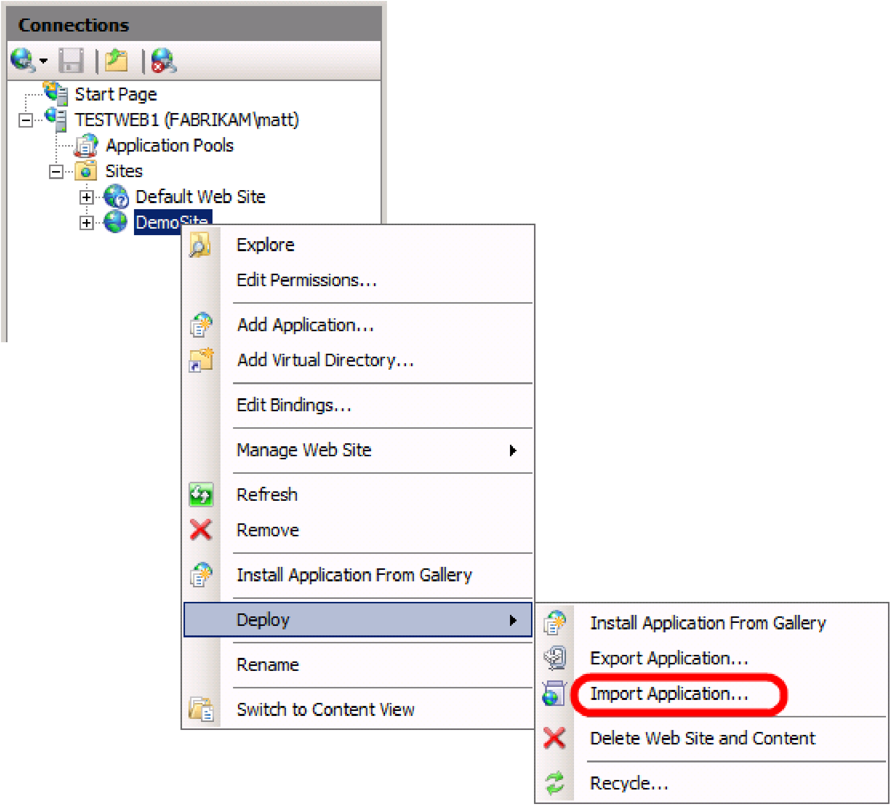
You can use the next procedure to import a web deployment package from the local file system into an IIS website. Before you perform this procedure, ensure that you have:

- Copied the web deployment package to the web server.
- Configured an IIS web server to host your application.

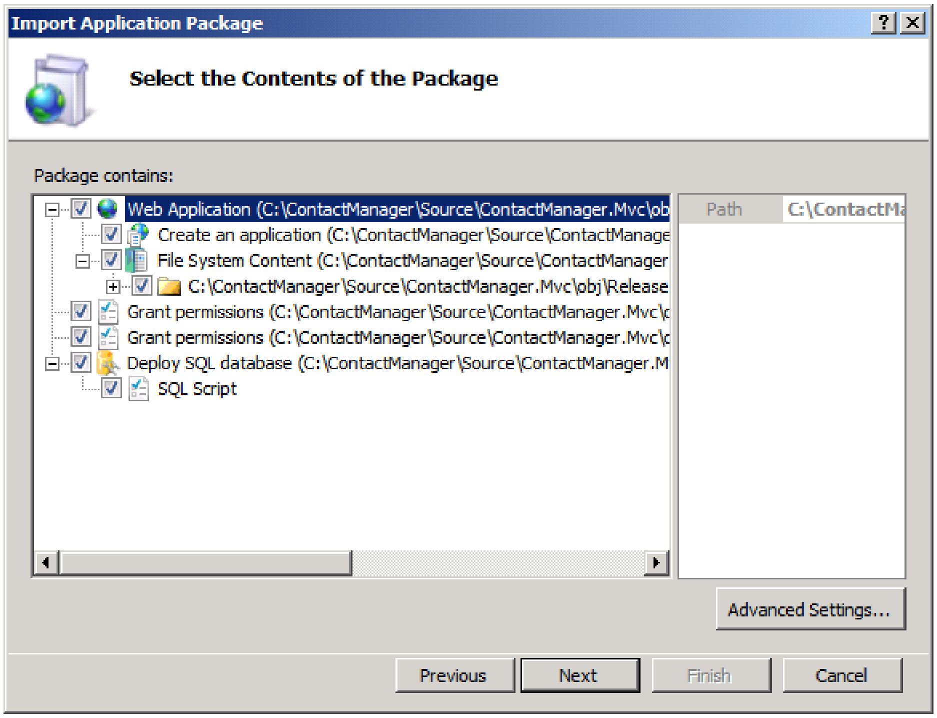
For more information on configuring an IIS web server to support web deployment packages, see Configure a Web Server for Web Deploy Publishing (Offline Deployment).

To import a web deployment package using IIS Manager

1. In IIS Manager, in the **Connections** pane, right-click your IIS website, point to **Deploy**, and then click **Import Application**.



2. In the Import Application Package Wizard, on the **Select the Package** page, browse to the location of your web deployment package, and then click **Next**.
3. On the **Select the Contents of the Package** page, clear any content that you don't require, and then click **Next**.



Import Application Package

Select the Contents of the Package

Package contains:

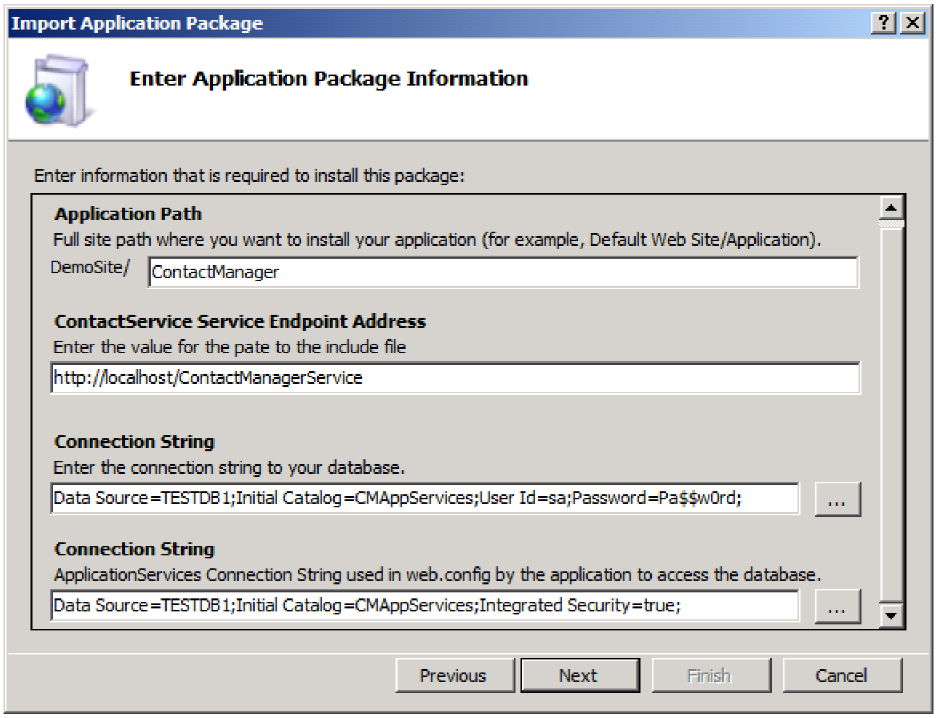
- ☑ Web Application (C:\ContactManager\Source\ContactManager.Mvc\ob
  - ☑ Create an application (C:\ContactManager\Source\ContactManage
  - ☑ File System Content (C:\ContactManager\Source\ContactManager
    - ☑ C:\ContactManager\Source\ContactManager.Mvc\obj\Release
- ☑ Grant permissions (C:\ContactManager\Source\ContactManager.Mvc\c
- ☑ Grant permissions (C:\ContactManager\Source\ContactManager.Mvc\c
- ☑ Deploy SQL database (C:\ContactManager\Source\ContactManager.M
  - ☑ SQL Script

Path: C:\ContactMa

Note: In a lot of cases, you may not want to import everything that comes with a web deployment package. For example, you may not want to allow Web Deploy to replace the associated database.

The **Grant permissions** entries set permissions on the destination file system to ensure that the application pool identity can access the physical folder that stores the website content. In addition, the anonymous authentication user is granted read permission to the folder to let the application serve Multipurpose Internet Mail Extensions (MIME) type files. If you prefer, you can remove these entries and configure permissions manually.

4. On the **Enter Application Package Information** page, provide the requested information.


Import Application Package
Enter Application Package Information
Previous
Next
Finish
Cancel

Enter information that is required to install this package:

Application Path
Full site path where you want to install your application (for example, Default Web Site/Application).
DemoSite/ ContactManager

ContactService Service Endpoint Address
Enter the value for the pate to the include file
http://localhost/ContactManagerService

Connection String
Enter the connection string to your database.
Data Source=TESTDB1;Initial Catalog=CMAppServices;User Id=sa;Password=Pa$$w0rd;

Connection String
ApplicationServices Connection String used in web.config by the application to access the database.
Data Source=TESTDB1;Initial Catalog=CMAppServices;Integrated Security=true;

When you create a web package, the WPP analyzes the configuration file for your application and detects any variables, like connection strings and service endpoints. In this case:

a. **Application Path** is the IIS path where you want to install your application. This setting is common to all deployment packages that the WPP creates.

b. **ContactService Service Endpoint Address** is the address that the application should use to communicate with the deployed WCF service. This setting corresponds to an entry in the *web.config* file.

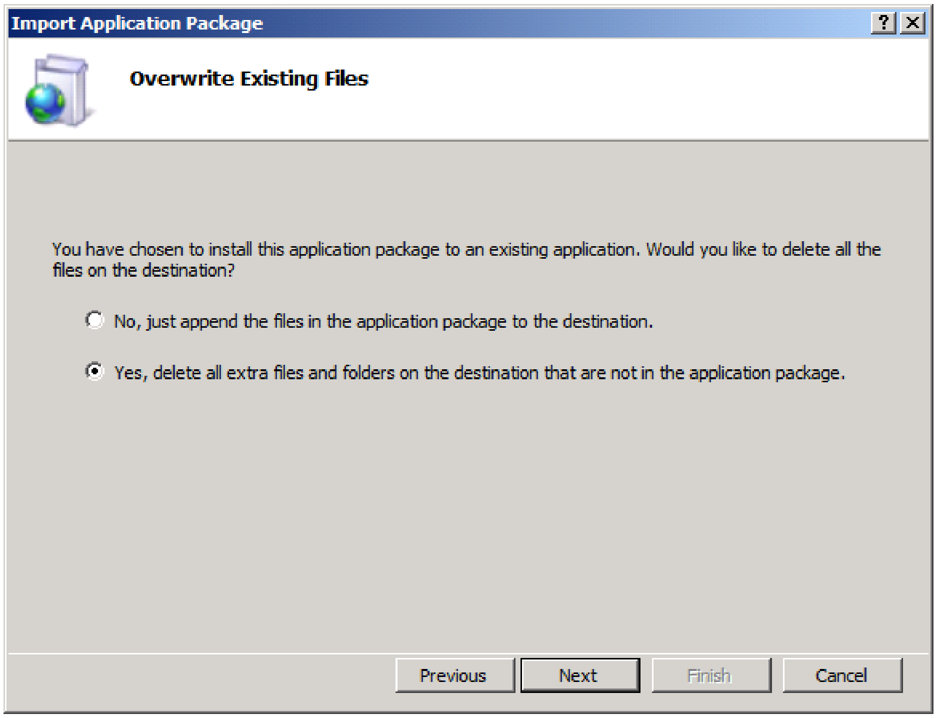
c. The first **Connection String** setting is the connection string that Web Deploy should use to deploy the database associated with the application (in this case an ASP.NET membership database). This setting corresponds to the setting on the **Package/Publish SQL** tab in Visual Studio.

d. The second **Connection String** setting is the connection string that your application will actually use to communicate with the database when it's up and running. This corresponds to a connection string entry in the *web.config* file.

> **Note:** For more information on where these parameters come from, see Configuring Parameters for Web Package Deployment.

5. Click **Next**.

6. If this is not the first time you've deployed the application to this website, you'll be prompted to specify whether you want to delete all existing content prior to installation. Choose the option that's appropriate for your requirements, and then click **Next**.

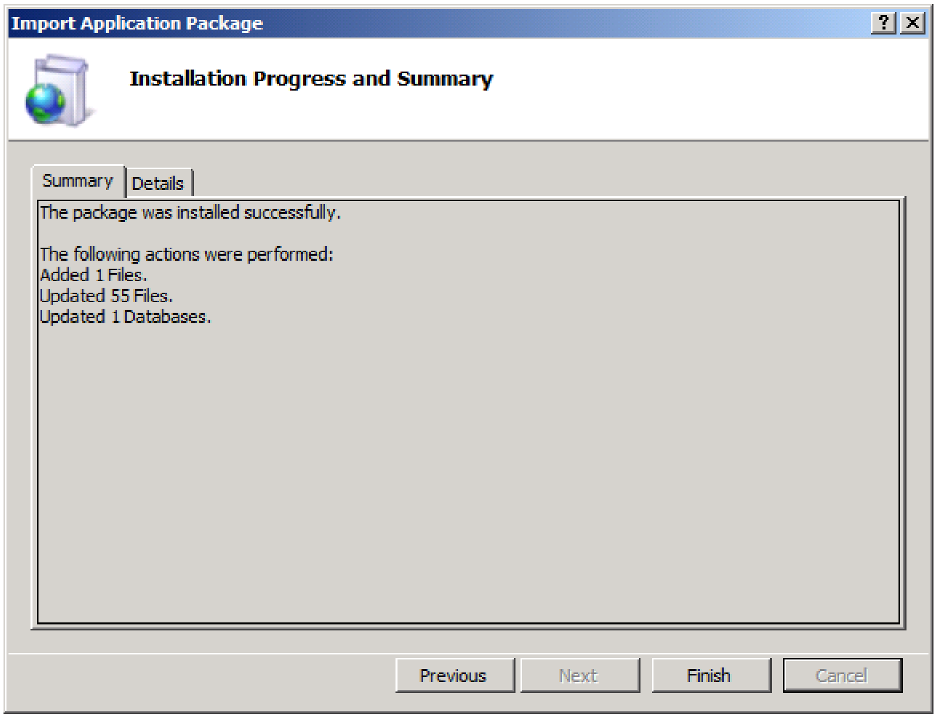


You have chosen to install this application package to an existing application. Would you like to delete all the files on the destination?

○ No, just append the files in the application package to the destination.

● Yes, delete all extra files and folders on the destination that are not in the application package.

7. When IIS has finished installing the package, click **Finish**.



```
The package was installed successfully.

The following actions were performed:
Added 1 Files.
Updated 55 Files.
Updated 1 Databases.
```

At this point, you've successfully published your web application to IIS.

Conclusion

This topic described how to import a web deployment package into an IIS website using IIS Manager. This approach to web application publishing is appropriate when security or infrastructure constraints make remote deployment impossible or undesirable.

Further Reading

For guidance on how to configure an IIS web server to support manually importing a web package, see Configure a Web Server for Web Deploy Publishing (Offline Deployment). For more general guidance on deploying web packages, see Walkthrough: Deploying a Web Application Project Using a Web Deployment Package (Part 1 of 4).

Configuring Server Environments for Web Deployment

This tutorial will show you how to set up server environments to support one-click, or automated, website deployment and publishing in various different scenarios. The tutorial includes topics to walk you through completing various tasks, like configuring a web server to support specific approaches to deployment and setting up a Web Farm Framework (WFF) server farm, together with scenario-based overviews that provide higher-level end-to-end guidance.

The tutorial uses the Fabrikam, Inc. deployment scenario described in Enterprise Web Deployment: Scenario Overview as a reference point for examples and network infrastructure.

How to Use This Tutorial

This tutorial includes these topics:

- Choosing the Right Approach to Web Deployment
- Scenario: Configuring a Test Environment for Web Deployment
- Scenario: Configuring a Staging Environment for Web Deployment
- Scenario: Configuring a Production Environment for Web Deployment
- Configuring a Web Server for Web Deploy Publishing (Remote Agent)
- Configuring a Web Server for Web Deploy Publishing (Web Deploy Handler)
- Configuring a Web Server for Web Deploy Publishing (Offline Deployment)
- Configuring a Database Server for Web Deploy Publishing
- Creating a Server Farm with the Web Farm Framework
- Configuring Deployment Properties for a Target Environment

The first topic, Choosing the Right Approach to Web Deployment, describes the main approaches you can use to publish web applications by using the Internet Information Services (IIS) Web Deployment Tool (Web Deploy) 2.0. It also identifies the scenarios that map to each approach. From here, each scenario topic provides a high-level overview of the tasks you need to complete and identifies the topics you'll need to work through to help you complete these tasks.

If you're using the split project file approach described in Understanding the Build Process to build and deploy your solution, the final topic, Configuring Deployment Properties for a Target Environment, describes how to configure environment-specific project files for deployment to different destination environments.

Key Technologies

This tutorial focuses on how to use these products and technologies to support web deployment:

- IIS 7.5
- Web Deploy 2.x
- WFF 2.x
- IIS Web Management Service (WMSvc)

The tutorial also touches on the use of Windows Server 2008 R2, SQL Server 2008 R2, ASP.NET 4.0, and ASP.NET MVC 3.

Choosing the Right Approach to Web Deployment

When you work with the Internet Information Services (IIS) Web Deployment Tool (Web Deploy) 2.0 or later, there are three main approaches you can use to get your packaged web applications onto a web server. You can either:

- Deploy the application from a remote location by targeting the *Web Deployment Agent Service* (also known as the "remote agent") on the destination server.
- Deploy the application from a remote location using Web Deploy On Demand (also known as the "temp agent").
- Deploy the application from a remote location by targeting the *IIS Web Deploy Handler* on the destination server.
- Deploy the application by manually copying the web package to the destination server and importing it through IIS Manager.

How you configure your destination web servers will depend on which approach to deployment you want to use. This topic will help you decide which approach to deployment is right for you.

Overview

This table shows the main advantages and disadvantages of each deployment approach, together with the scenarios that most typically suit each approach.

Approach	Advantages	Disadvantages	Typical Scenarios
Remote Agent	It is easy to set up. It is suitable for regular updates to web applications and content.	The user must be an administrator on the target server. the user can't supply alternative credentials.	Development environments. Test environments.
Temp Agent	There is no need to install Web Deploy on the target computer.	The user must be an administrator on the target server.	Development environments. Test environments.

	The latest version of Web Deploy is automatically used.	The user can't supply alternative credentials.	
Web Deploy Handler	Non-administrator users can deploy content. It is suitable for regular updates to web applications and content.	It is a lot more complex to set up.	Staging environments. Intranet production environments. Hosted environments.
Offline Deployment	It is very easy to set up. It is suitable for isolated environments.	The server administrator must manually copy and import the web package every time.	Internet-facing production environments. Isolated network environments.

Using the Remote Agent

When you install Web Deploy using the default settings on a destination server, the Web Deployment Agent Service (the "remote agent") is automatically installed and started. By default, the remote agent exposes an HTTP endpoint at this address:

```
http://[server]/MSDEPLOYAGENTSERVICE
```

Note: You can replace [*server*] with the machine name of your web server, an IP address for your web server, or a hostname that resolves to your web server.

Server administrators can deploy web packages from a remote location, like a developer machine or a build server, by specifying this endpoint address. For example, suppose Matt Hink at Fabrikam, Inc. has built the ContactManager.Mvc web application project on his developer machine. The build process generates a web package, together with a *.deploy.cmd* file that contains the Web Deploy commands required to install the package. If Matt is a server administrator on the TESTWEB1 server, he can deploy the web application to the test web server by running this command on his developer machine:

```
ContactManager.Mvc.deploy.cmd /y /m:http://TESTWEB1/MSDEPLOYAGENTSERVICE a/:NTLM
```

In actual fact, the Web Deploy executable can infer the endpoint address of the remote agent if you provide the machine name, so Matt only needs to type this:

```
ContactManager.Mvc.deploy.cmd /y /m:TESTWEB1 /a:NTLM
```

Note: For more information on Web Deploy command-line syntax and *.deploy.cmd* files, see How to: Install a Deployment Package Using the deploy.cmd File.

The remote agent offers a straightforward way to deploy content from a remote location, and this approach can work well with one-click or automated deployment. However, the user who runs the deployment command must also be either a domain administrator or a member of the local administrators group on the destination server. In addition, the remote agent doesn't support basic authentication, so you can't pass alternative credentials on the command line.

The remote agent provides a useful approach to deployment in development or test scenarios, where it's not uncommon for developers to have full administrator control over a test server environment, and applications are typically rebuilt and redeployed very frequently. However, this approach is usually less acceptable for staging or production environments.

For an end-to-end example of a scenario that uses the remote agent approach, see Scenario: Configuring a Test Environment for Web Deployment.

Using the Temp Agent

The temp agent approach to deployment is similar to the remote agent approach. However, in contrast to the remote agent approach, you don't need to install Web Deploy on the destination web server. Instead, when you perform the deployment, Web Deploy will install a temporary version of the web deployment agent service on the destination server and will use this to deploy your content to IIS. When the deployment is complete, all temporary files are removed.

If you want to use the temp agent provider setting, add the **/g** flag to your deployment command:

```
ContactManager.Mvc.deploy.cmd /y /m:TESTWEB1 /g:true
```

Note: You can't use the temp agent if the web deployment agent service is installed on the destination computer, even if the service is not running.

The advantage of this approach is that you don't need to maintain installations of Web Deploy on your destination servers. Furthermore, you don't need to ensure that the source and destination computers are running the same version of Web Deploy. However, this approach suffers from the same principal limitations as the remote agent approach, namely that you must be a local administrator on the destination server in order to deploy content, and only NTLM authentication is supported. The temp agent approach also requires a lot more initial configuration of the destination environment.

For more information on using the temp agent, see How to: Install a Deployment Package Using the deploy.cmd File and Web Deploy On Demand.

Using the Web Deploy Handler

For IIS 7 onwards, Web Deploy offers an alternative deployment approach through the IIS Web Deploy Handler. The Web Deploy Handler is closely integrated with the IIS Web Management Service (WMSvc), which is designed to allow users to manage IIS websites from remote locations.

By default, the remote agent exposes an HTTP endpoint at this address:

```
https://[server]:8172/MSDeploy.axd
```

Note: You can replace [*server*] with the machine name of your web server, an IP address for your web server, or a hostname that resolves to your web server.

The big advantage of the Web Deploy Handler over the remote agent, and the temp agent, is that you can configure IIS to allow non-administrator users to deploy applications and content to specific IIS websites. The Web Deploy Handler also supports basic authentication, so you can provide alternative credentials as parameters in your Web Deploy commands. The major drawback is that the Web Deploy Handler is initially a lot more complicated to set up and configure.

In the case of non-administrator users, the Web Management Service (WMSvc) will only allow the user to connect to IIS using a site-level connection, rather than a server-level connection. To access a particular site, you can include a site-specific query string in the endpoint address:

```
https://[server]:8172/MSDeploy.axd?site=DemoSite
```

For example, suppose a build process is configured to automatically deploy a web application to a staging environment after every successful build. If you used the remote agent approach, you'd need to make the build process identity an administrator on your destination servers. In contrast, using the Web Deploy Handler approach you can give a non-administrator user—**FABRIKAM\stagingdeployer** in this case—permission to a specific IIS website only, and the build process can provide these credentials to deploy the web package.

```
msdeploy.exe
  -source:package='…\ContactManager.Mvc.zip'
  -dest:auto,
        computerName='https://STAGEWEB1:8172/MSDeploy.axd?site=DemoSite',
        userName='FABRIKAM\stagingdeployer',
        password='Pa$$w0rd',
        authtype='Basic',
  -verb:sync
  -setParamFile:"…\ContactManager.Mvc.SetParameters.xml"
  -allowUntrusted
```

Note: For more information on Web Deploy command-line operations and syntax, see Web Deploy Command Line Reference. For more information on using the *.deploy.cmd* file, see How to: Install a Deployment Package Using the deploy.cmd File.

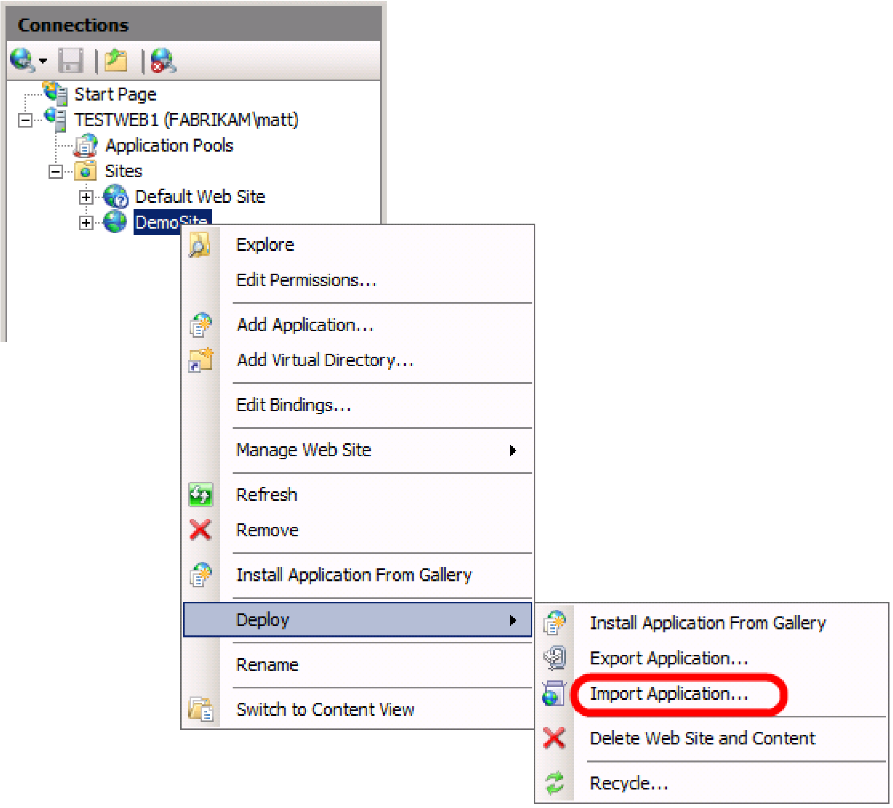
The Web Deploy Handler provides a useful approach to deployment in staging environments, hosted environments, and intranet-based production environments, where remote access to the server is available but administrator credentials are not.

For an end-to-end example of a scenario that uses the Web Deploy Handler approach, see Scenario: Configuring a Staging Environment for Web Deployment.

Using Offline Deployment

In some cases, it's not possible or practical to deploy applications and content to an IIS website from a remote location. For example, the source and destination computers may be in isolated networks or network segments, or firewall policy may not permit remote access.

In scenarios like these, you can still use the packaging and publishing capabilities of Web Deploy; you just can't use them from a remote location. Instead, an administrator on the destination server must copy the web package onto the server and import it through IIS Manager.



The offline deployment approach is typically useful in Internet-facing production environments, where servers in a perimeter network may have restricted connectivity with computers in the internal network.

For an end-to-end example of a scenario that uses the offline deployment approach, see Scenario: Configuring a Production Environment for Web Deployment.

Further Reading

For more information on Web Deploy command-line operations and syntax, see Web Deploy Command Line Reference. For more information on using the *.deploy.cmd* file, see How to: Install a Deployment Package Using the deploy.cmd File.

For more general guidance on the different ways in which you can deploy web packages from a remote computer, see Using Web Deploy Remotely. For more information on using Web Deploy On Demand, see Web Deploy On Demand.

Scenario: Configuring a Test Environment for Web Deployment

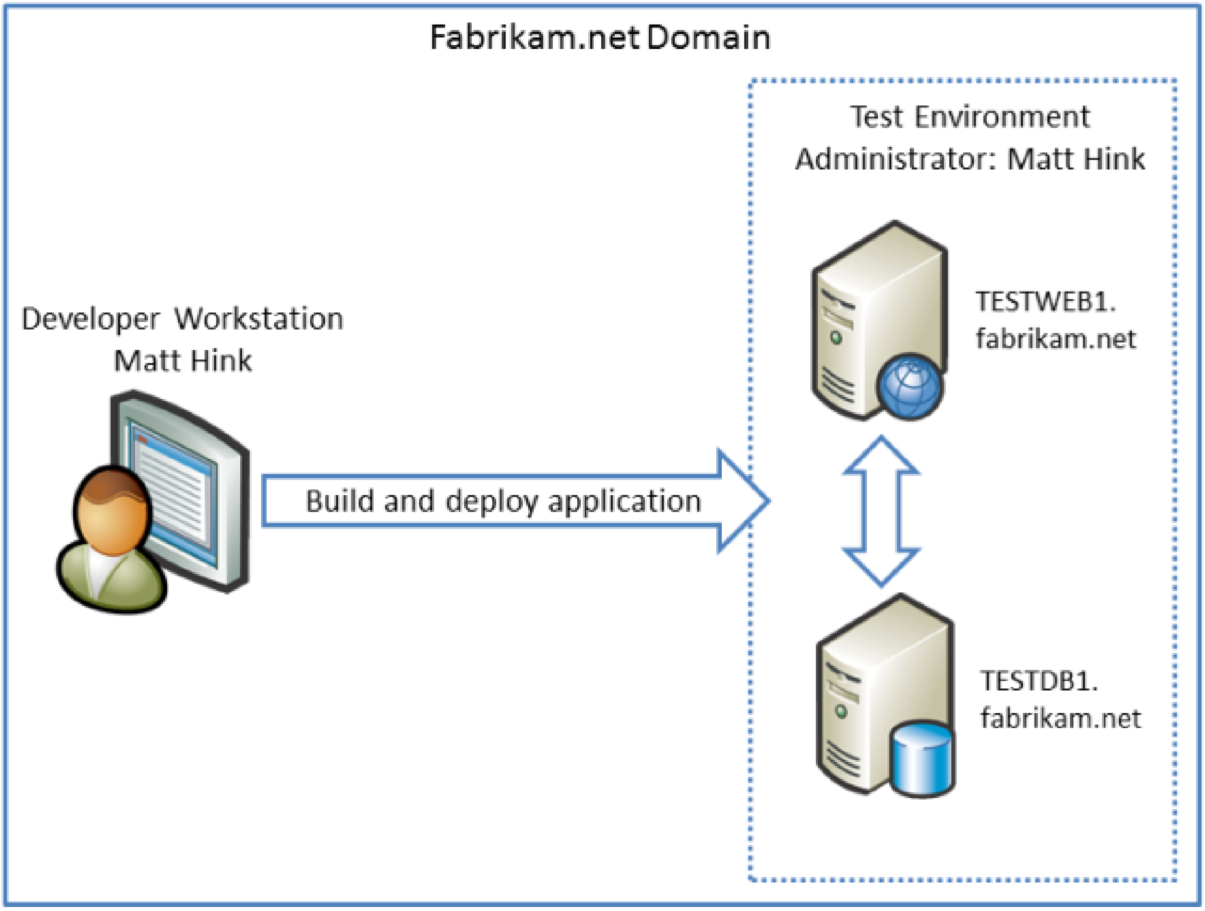
This topic describes a typical web deployment scenario for developer or test environments and explains the tasks you need to complete in order to set up a similar environment.

Scenario Overview

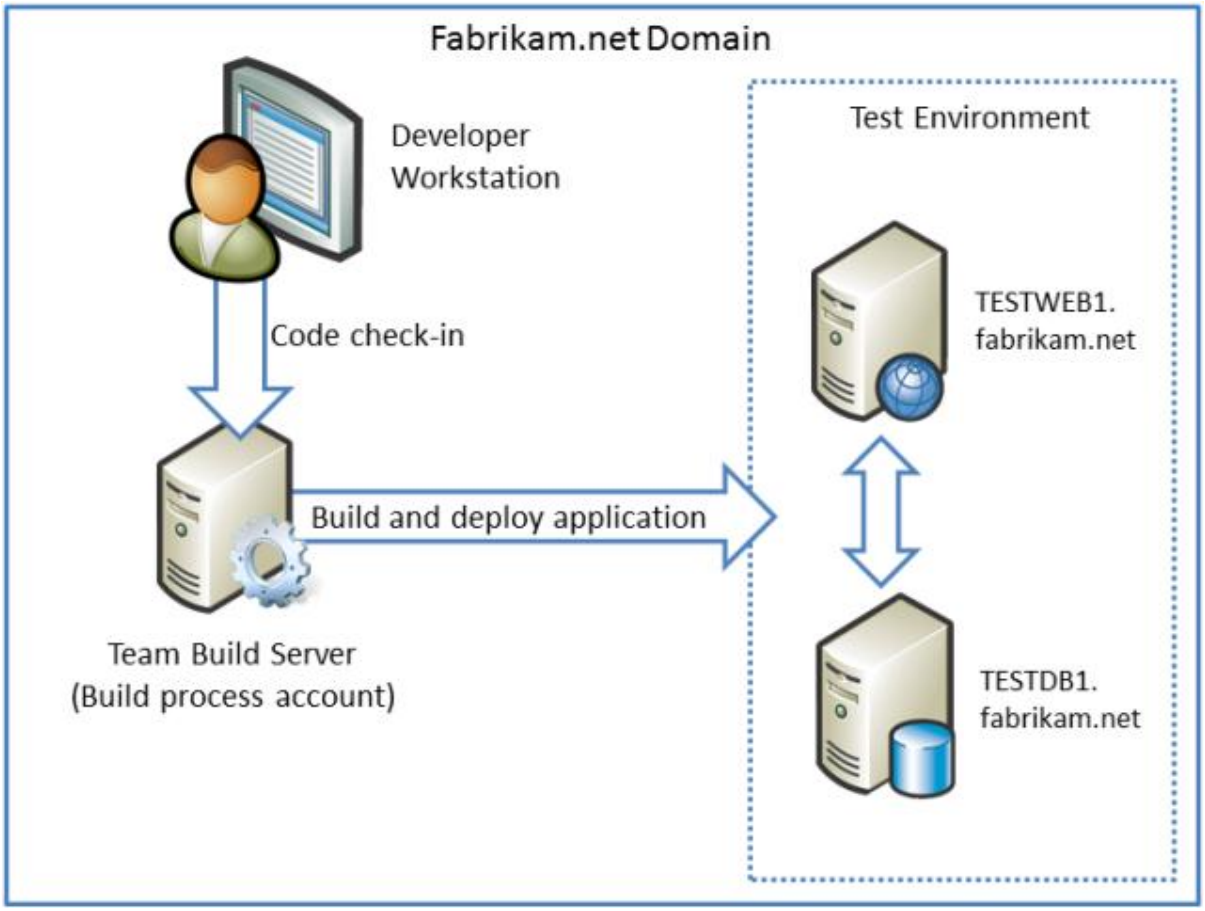
When developers work on web applications, they're often given access to a server environment that they can use to test changes to their applications in a realistic setting. This kind of development or test environment typically has these characteristics:

- The environment consists of a single web server and a single database server.
- The developers usually have administrator privileges on the servers, to let them configure the environment to the requirements of their applications.
- Changes to applications are deployed on a frequent basis, so the environment needs to support single-step or automated deployment.

For example, in our tutorial scenario, Matt Hink is a developer at Fabrikam, Inc. Matt is working on the Contact Manager solution and regularly needs to deploy changes to a test environment. Matt is an administrator on the test web server and the test database server. Initially, Matt needs to be able to deploy the solution to the test environment directly.



As work progresses and more developers join the team, the Contact Manager solution is configured for continuous integration (CI) in Team Foundation Server (TFS). Whenever a developer checks in content, Team Build should build the solution, run any unit tests, and automatically deploy the solution to the test environment.



Solution Overview

The test environment needs to support single-step or automated deployment from a remote computer, so you have a choice of two main approaches. You can:

- Configure the test web server to support deployment using the Web Deployment Agent Service (the "remote agent").
- Configure the test web server to support deployment using the Web Deploy handler.

Note: You could also use Web Deploy On Demand (the "temp agent"). This is similar to the remote agent approach in terms of requirements and constraints.

In this case, the developers have administrator privileges on the destination servers, and the test environment is not subject to strict security constraints, so the logical choice is to configure the test web server to support deployment using the remote agent. This is less complex and requires less initial configuration than the Web Deploy Handler approach. You'll also need to configure your database server to support remote access and deployment.

These topics provide all the information you need in order to complete these tasks:

- Configure a Web Server for Web Deploy Publishing (Remote Agent). This topic describes how to build a web server that supports Web Deploy publishing, using the remote agent approach, starting from a clean Windows Server 2008 R2 build.
- Configure a Database Server for Web Deploy Publishing. This topic describes how to configure a database server to support remote access and deployment, starting from a default installation of SQL Server 2008 R2.

Further Reading

For guidance on configuring a typical staging environment, see Scenario: Configuring a Staging Environment for Web Deployment. For guidance on configuring a typical production environment, see Scenario: Configuring a Production Environment for Web Deployment.

Scenario: Configuring a Staging Environment for Web Deployment

This topic describes a typical web deployment scenario for a staging environment and explains the tasks you need to complete in order to set up a similar environment.

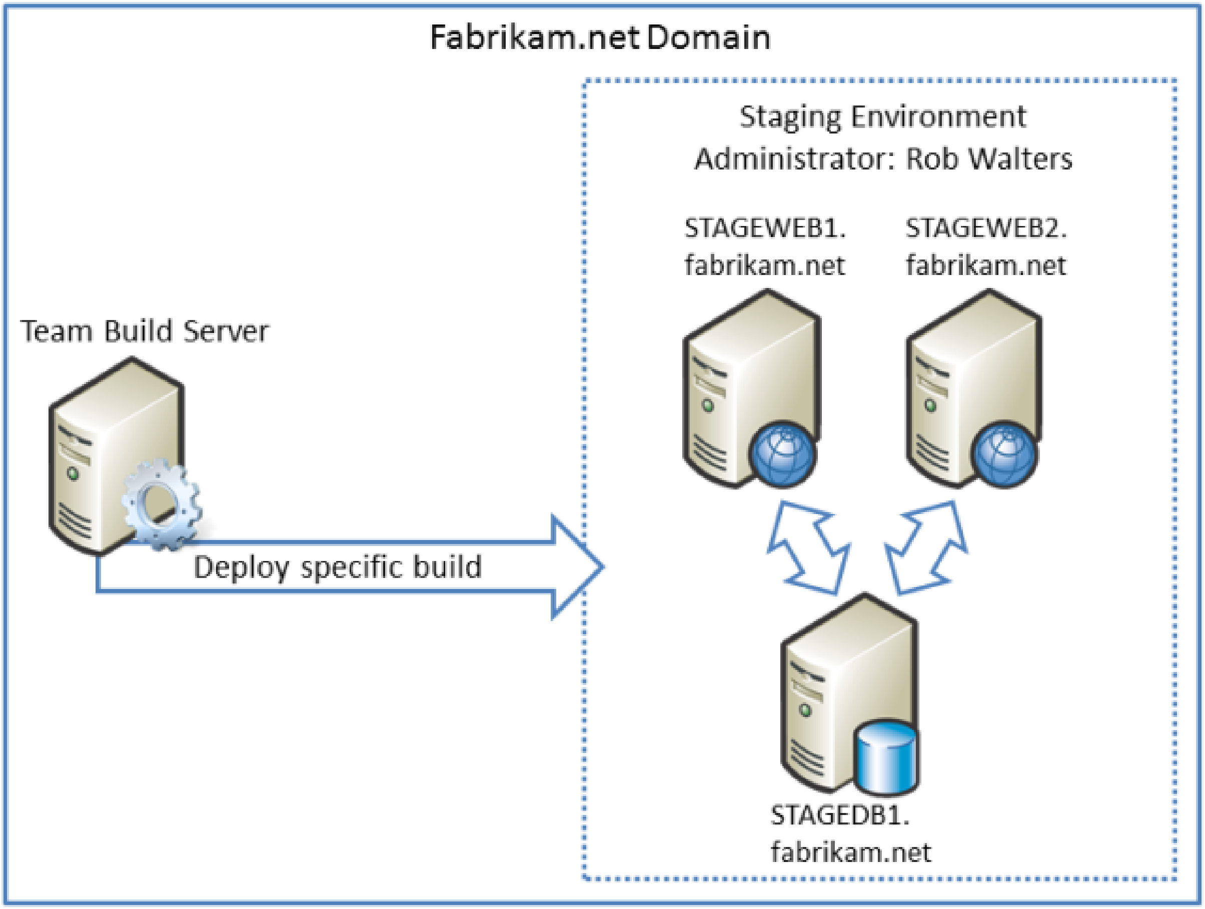
Scenario Overview

Lots of organizations use staging environments to preview updates to web applications or websites. This gives people within the organization a chance to explore and review new functionality or content before the site "goes live," or in other words is deployed to a production environment. The staging environment is designed to replicate the production environment as closely as possible, in order to provide a realistic preview. This kind of staging environment typically has these characteristics:

- The environment consists of multiple load-balanced web servers and one or more database servers, often with failover clustering and database mirroring.
- Applications may be deployed manually by a development team or automatically by a Team Build server.
- The users or process accounts that deploy applications are unlikely to have administrator privileges on the staging servers.
- Changes to applications are deployed on a frequent basis, so the environment needs to support single-step or automated deployment.

Note: Scaling out a database deployment across multiple servers is beyond the scope of this tutorial. For more information on this area, please consult SQL Server Books Online.

For example, in our tutorial scenario, Team Foundation Server (TFS) manages the Contact Manager solution. The TFS administrator, Rob Walters, has created a build definition that lets developers trigger a deployment to the staging environment as required.



Note that in most cases, you won't necessarily want to deploy the latest build to the staging environment. Instead, you're a lot more likely to want to deploy a specific build that has already undergone validation and verification in the test environment.

Solution Overview

In this scenario, you can deduce these facts from an analysis of the deployment requirements:

- The user or process account that performs the deployment won’t have administrator privileges on the staging servers, so the staging web servers must support non-administrator deployment. As such, you'll need to configure the staging web servers to use the Web Deploy Handler rather than the remote agent.
- The staging environment includes multiple web servers, but it needs to support one-click or automated deployment, so you'll need to use the Web Farm Framework (WFF) to create a server farm. Using this approach, you can deploy an application to one web server (the primary server), and WFF will replicate the deployment on all the other web servers in the staging environment.
- The user or process account that performs the deployment must have permissions to create databases. As such, you'll need to add the account to the **dbcreator** server role on the database

server, in addition to configuring the database server to support remote access and deployment.

These topics provide all the information you need in order to complete these tasks:

- Create a Server Farm with the Web Farm Framework. This topic describes how to create and configure a server farm using WFF, so that web platform products and components, configuration settings, and websites and applications are replicated across multiple load-balanced web servers.
- Configure a Web Server for Web Deploy Publishing (Web Deploy Handler). This topic describes how to build a web server that supports Web Deploy publishing, using the remote agent approach, starting from a clean Windows Server 2008 R2 build.
- Configure a Database Server for Web Deploy Publishing. This topic describes how to configure a database server to support remote access and deployment, starting from a default installation of SQL Server 2008 R2.

Further Reading

For guidance on configuring a typical developer test environment, see Scenario: Configuring a Test Environment for Web Deployment. For guidance on configuring a typical production environment, see Scenario: Configuring a Production Environment for Web Deployment.

Scenario: Configuring a Production Environment for Web Deployment

This topic describes a typical web deployment scenario for a production environment and explains the tasks you need to complete in order to set up a similar environment.

Scenario Overview

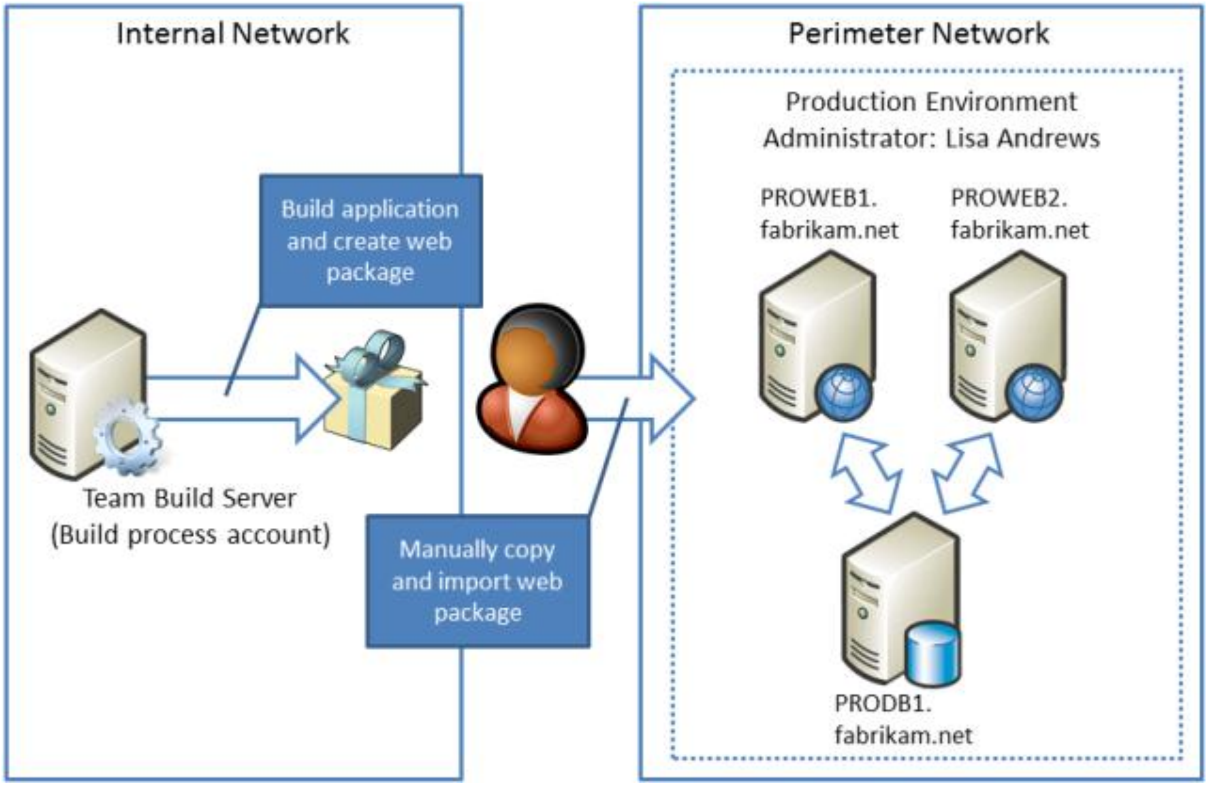
The production environment is the final destination for a web application or a website. By this point, your application has been through testing, has been deployed to a staging environment, and is ready to "go live." The characteristics of a production environment can vary widely according to the nature and purpose of your web content, the size of your organization, your target audience, and lots of other factors. In an enterprise-scale scenario, the production environment may have these characteristics:

- The environment consists of multiple load-balanced web servers and one or more database servers, often with failover clustering and database mirroring.
- If the environment is Internet-facing, it's likely to be segregated from your internal network. It may be on a different subnet in a perimeter network, it may be on a different domain, and it may be on an entirely different network infrastructure.
- Developers and build server process accounts are highly unlikely to have administrator privileges on the production servers.

- Changes to applications are deployed on a less frequent basis than test or staging deployments.

Note: Scaling out a database deployment across multiple servers is beyond the scope of this tutorial. For more information on this area, please consult SQL Server Books Online.

For example, in our tutorial scenario, a Team Build server includes build definitions that let users build the Contact Manager solution and deploy it to a staging environment in a single step. When the application is ready to be deployed to production, due to the constraints imposed by security requirements and the network infrastructure, the production environment administrator must manually copy the web package onto a production web server and import it through Internet Information Services (IIS) Manager.



Solution Overview

In this scenario, you can deduce these facts from an analysis of the deployment requirements:

- Due to security restrictions and the network configuration, you can't configure the production environment to support one-click or automated deployment. Offline deployment is the only viable approach in this scenario.
- The production environment includes multiple web servers, so you can use the Web Farm Framework (WFF) to create a server farm. Using this approach, the administrator only needs to import the application onto one web server (the primary server), and WFF will replicate the deployment on all the other web servers in the production environment.

These topics provide all the information you need in order to complete these tasks:

- Create a Server Farm with the Web Farm Framework. This topic describes how to create and configure a server farm using WFF, so that web platform products and components, configuration settings, and websites and applications are replicated across multiple load-balanced web servers.
- Configure a Web Server for Web Deploy Publishing (Offline Deployment). This topic describes how to build a web server that lets administrators import and deploy web packages manually, starting from a clean Windows Server 2008 R2 build.
- Configure a Database Server for Web Deploy Publishing. This topic describes how to configure a database server to support remote access and deployment, starting from a default installation of SQL Server 2008 R2.

Further Reading

For guidance on configuring a typical developer test environment, see Scenario: Configuring a Test Environment for Web Deployment. For guidance on configuring a typical staging environment, see Scenario: Configuring a Staging Environment for Web Deployment.

Configuring a Web Server for Web Deploy Publishing (Remote Agent)

This topic describes how to configure an Internet Information Services (IIS) web server to support web publishing and deployment using the IIS Web Deployment Tool (Web Deploy) Remote Agent Service.

When you work with Web Deploy 2.0 or later, there are three main approaches you can use to get your applications or sites onto a web server. You can:

- Use the *Web Deploy Remote Agent Service*. This approach requires less configuration of the web server, but you need to provide the credentials of a local server administrator in order to deploy anything to the server.
- Use the *Web Deploy Handler*. This approach is a lot more complex and requires more initial effort to set up the web server. However, when you use this approach, you can configure IIS to allow non-administrator users to perform the deployment. The Web Deploy Handler is only available in IIS version 7 or later.
- Use *offline deployment*. This approach requires the least configuration of the web server, but a server administrator must manually copy the web package onto the server and import it through IIS Manager.

For more information on the key features, advantages, and disadvantages of these approaches, see Choosing the Right Approach to Web Deployment.

Is the Web Deploy Remote Agent the Right Approach for You?

Yes, if the user who will deploy the content can supply the credentials of an administrator on the destination server. This approach is often desirable in these types of scenarios:

- Development or test environments, where the developer has full control over the destination web server and database server.
- Smaller organizations in which a single user or a small group of users has control over the entire application lifecycle.

In lots of larger organizations, and particularly for staging or production environments, it's often not realistic to give users administrator rights on web servers. In the case of hosted web servers, this is especially unlikely to be the case. In addition, if you're planning to automate deployment from a build server, you may not want to use administrator credentials for the deployment process. In these scenarios, configuring your web servers to support deployment using the Web Deploy Handler may provide a more satisfactory choice.

Task Overview

To configure the web server to accept and deploy web packages from a remote computer using the Web Deploy Remote Agent approach, you'll need to:

- Install IIS 7.5 and the IIS 7 recommended configuration.
- Install Web Deploy 2.1 or later.
- Create an IIS website to host the deployed content.
- Ensure that the Web Deployment Agent Service is running.

To host the sample solution specifically, you'll also need to:

- Install the .NET Framework 4.0.
- Install ASP.NET MVC 3.

This topic will show you how to perform each of these procedures. The tasks and walkthroughs in this topic assume that you're starting with a clean server build running Windows Server 2008 R2. Before you continue, ensure that:

- Windows Server 2008 R2 Service Pack 1 and all available updates are installed.
- The server is domain-joined.
- The server has a static IP address.

Note: For more information on joining computers to a domain, see Joining Computers to the Domain and Logging On. For more information on configuring static IP addresses, see Configure a Static IP Address.

Install Products and Components

This section will guide you through installing the required products and components on the web server. Before you begin, a good practice is to run Windows Update to ensure that your server is fully up to date.

In this case, you need to install these things:

- **IIS 7 Recommended Configuration**. This enables the **Web Server (IIS)** role on your web server and installs the set of IIS modules and components that you need in order to host an ASP.NET application.
- **.NET Framework 4.0**. This is required to run applications that were built on this version of the .NET Framework.
- **Web Deployment Tool 2.1 or later**. This installs Web Deploy (and its underlying executable, MSDeploy.exe) on your server. As part of this process, it installs and starts the Web Deployment Agent Service. This service lets you deploy web packages from a remote computer.
- **ASP.NET MVC 3**. This installs the assemblies you need to run MVC 3 applications.

Note: This walkthrough describes the use of the Web Platform Installer to install and configure the required components. Although you don't have to use the Web Platform Installer, it simplifies the installation process by automatically detecting dependencies and ensuring that you always get the latest product versions. For more information, see Microsoft Web Platform Installer 3.0.

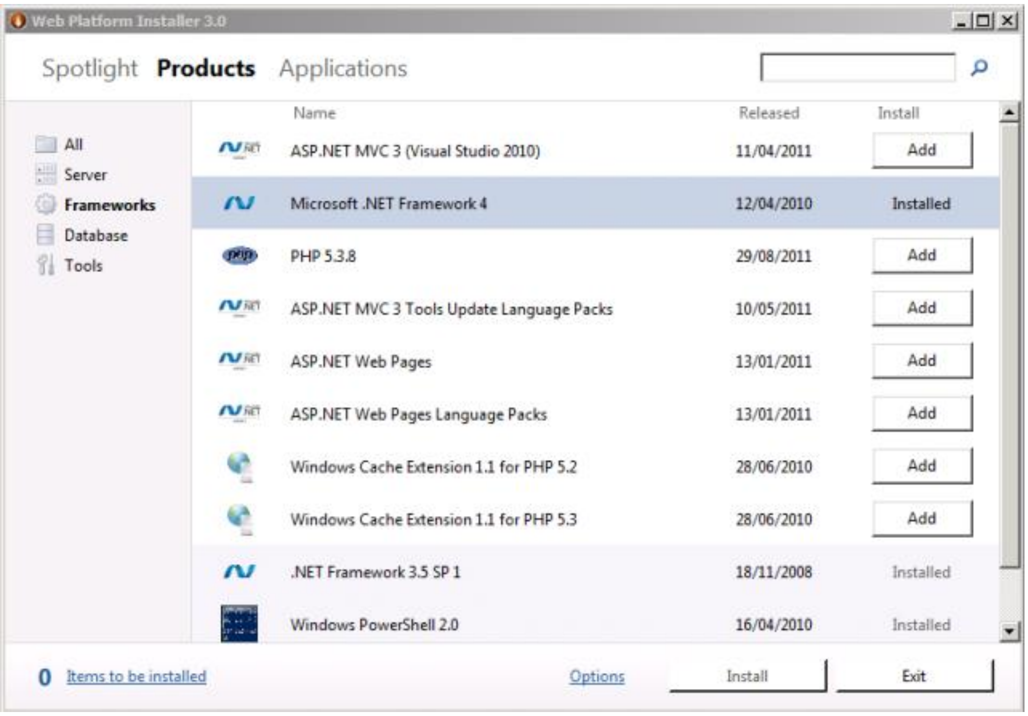
To install the required products and components

1. Download and install the Web Platform Installer.
2. When installation is complete, the Web Platform Installer will launch automatically.

   Note: You can now launch the Web Platform Installer at any time from the **Start** menu. To do this, on the **Start** menu, click **All Programs**, and then click **Microsoft Web Platform Installer**.

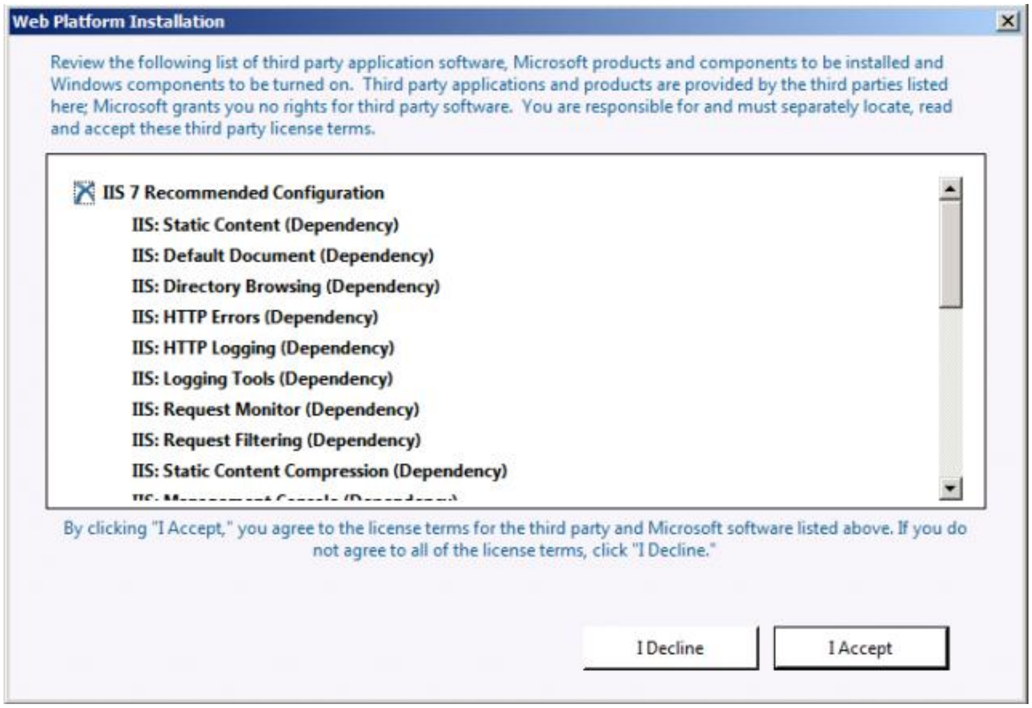
3. At the top of the **Web Platform Installer 3.0** window, click **Products**.
4. On the left side of the window, in the navigation pane, click **Frameworks**.
5. In the **Microsoft .NET Framework 4** row, if the .NET Framework is not already installed, click **Add**.

   Note: You may have already installed the .NET Framework 4.0 through Windows Update. If a product or component is already installed, the Web Platform Installer will indicate this by replacing the **Add** button with the text **Installed**.



Name	Released	Install
ASP.NET MVC 3 (Visual Studio 2010)	11/04/2011	Add
Microsoft .NET Framework 4	12/04/2010	Installed
PHP 5.3.8	29/08/2011	Add
ASP.NET MVC 3 Tools Update Language Packs	10/05/2011	Add
ASP.NET Web Pages	13/01/2011	Add
ASP.NET Web Pages Language Packs	13/01/2011	Add
Windows Cache Extension 1.1 for PHP 5.2	28/06/2010	Add
Windows Cache Extension 1.1 for PHP 5.3	28/06/2010	Add
.NET Framework 3.5 SP 1	18/11/2008	Installed
Windows PowerShell 2.0	16/04/2010	Installed

6. In the **ASP.NET MVC 3 (Visual Studio 2010)** row, click **Add**.
7. In the navigation pane, click **Server**.
8. In the **IIS 7 Recommended Configuration** row, click **Add**.
9. In the **Web Deployment Tool 2.1** row, click **Add**.
10. Click **Install**. The Web Platform Installer will show you a list of products—together with any associated dependencies—to be installed and will prompt you to accept the license terms.



Web Platform Installation

Review the following list of third party application software, Microsoft products and components to be installed and Windows components to be turned on. Third party applications and products are provided by the third parties listed here; Microsoft grants you no rights for third party software. You are responsible for and must separately locate, read and accept these third party license terms.

IIS 7 Recommended Configuration
IIS: Static Content (Dependency)
IIS: Default Document (Dependency)
IIS: Directory Browsing (Dependency)
IIS: HTTP Errors (Dependency)
IIS: HTTP Logging (Dependency)
IIS: Logging Tools (Dependency)
IIS: Request Monitor (Dependency)
IIS: Request Filtering (Dependency)
IIS: Static Content Compression (Dependency)
IIS: Management Console (Dependency)

By clicking "I Accept," you agree to the license terms for the third party and Microsoft software listed above. If you do not agree to all of the license terms, click "I Decline."

11. Review the license terms, and if you consent to the terms, click **I Accept**.
12. When the installation is complete, click **Finish**, and then close the **Web Platform Installer 3.0** window.

If you installed the .NET Framework 4.0 before you installed IIS, you'll need to run the ASP.NET IIS Registration Tool (aspnet_regiis.exe) to register the latest version of ASP.NET with IIS. If you don't do this, you'll find that IIS will serve static content (like HTML files) without any problems, but it will return **HTTP Error 404.0 – Not Found** when you attempt to browse to ASP.NET content. You can use this procedure to ensure that ASP.NET 4.0 is registered.

To register ASP.NET 4.0 with IIS

1. Click **Start**, and then type **Command Prompt**.
2. In the search results, right-click **Command Prompt**, and then click **Run as administrator**.
3. In the Command Prompt window, navigate to the **%WINDIR%\Microsoft.NET\Framework\v4.0.30319** directory.
4. Type this command, and then press Enter:

```
aspnet_regiis -iru
```

5. If you plan to host 64-bit web applications at any point, you should also register the 64-bit version of ASP.NET with IIS. To do this, in the Command Prompt window, navigate to the **%WINDIR%\Microsoft.NET\Framework64\v4.0.30319** directory.
6. Type this command, and then press Enter:

```
aspnet_regiis -iru
```

As a good practice, use Windows Update again at this point to download and install any available updates for the new products and components you've installed.

Configure the IIS Website

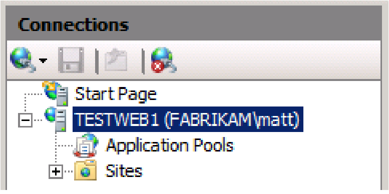
Before you can deploy web content to your server, you need to create and configure an IIS website to host the content. Web Deploy can only deploy web packages to an existing IIS website; it can't create the website for you. At a high level, you'll need to complete these tasks:

- Create a folder on the file system to host your content.
- Create an IIS website to serve the content, and associate it with the local folder.
- Grant read permissions to the application pool identity on the local folder.

Although there's nothing stopping you from deploying content to the default website in IIS, this approach is not recommended for anything other than test or demonstration scenarios. To simulate a production environment, you should create a new IIS website with settings that are specific to the requirements of your application.

To create and configure an IIS website

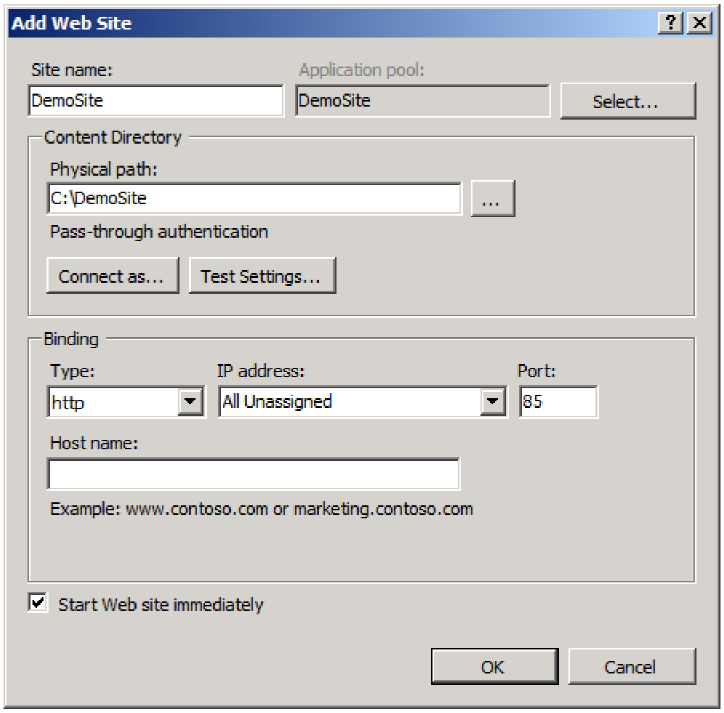
1. On the local file system, create a folder to store your content (for example, **C:\DemoSite**).
2. On the **Start** menu, point to **Administrative Tools**, and then click **Internet Information Services (IIS) Manager**.
3. In IIS Manager, in the **Connections** pane, expand the server node (for example, **TESTWEB1**).



4. Right-click the **Sites** node, and then click **Add Web Site**.
5. In the **Site name** box, type a name for the IIS website (for example, **DemoSite**).
6. In the **Physical path** box, type (or browse to) the path to your local folder (for example, **C:\DemoSite**).
7. In the **Port** box, type the port number on which you want to host the website (for example, **85**).

> **Note:** The standard port numbers are 80 for HTTP and 443 for HTTPS. However, if you host this website on port 80, you’ll need to stop the default website before you can access your site.

8. Leave the **Host name** box blank, unless you want to configure a Domain Name System (DNS) record for the website, and then click **OK**.



Site name: DemoSite

Application pool: DemoSite

Physical path: C:\DemoSite

Type: http

IP address: All Unassigned

Port: 85

Host name:

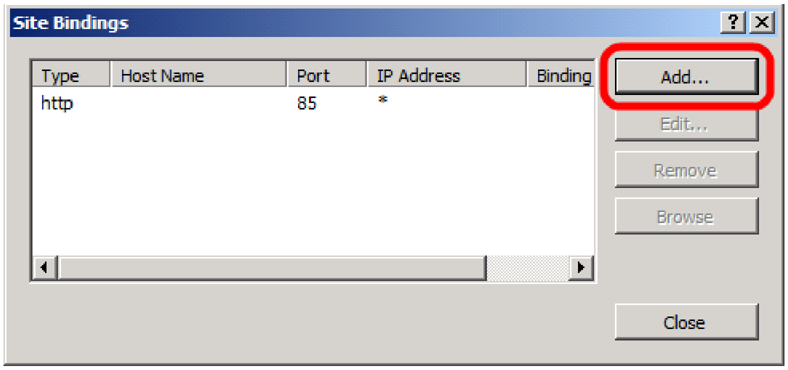
Example: www.contoso.com or marketing.contoso.com

☑ Start Web site immediately

Note: In a production environment, you'll likely want to host your website on port 80 and configure a host header, together with matching DNS records. For more information on configuring host headers in IIS 7, see Configure a Host Header for a Web Site (IIS 7). For more information on the DNS Server role in Windows Server 2008 R2, see DNS Server Overview and DNS Server.

9. In the **Actions** pane, under **Edit Site**, click **Bindings**.

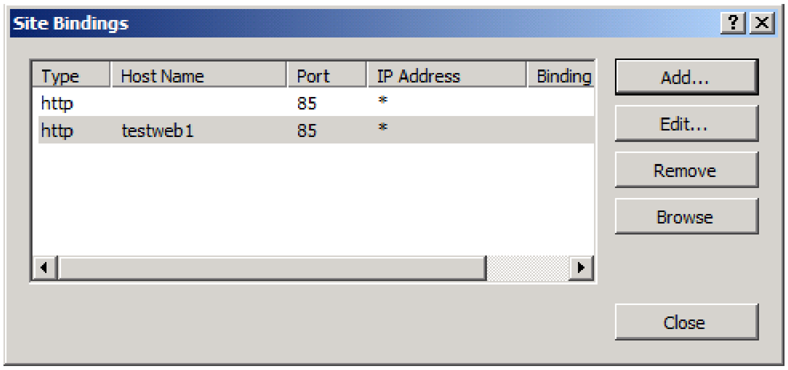
10. In the **Site Bindings** dialog box, click **Add**.



Type	Host Name	Port	IP Address	Binding
http		85	*	

11. In the **Add Site Binding** dialog box, set the **IP address** and **Port** to match your existing site configuration.

12. In the **Host name** box, type the name of your web server (for example, **TESTWEB1**), and then click **OK**.



Site Bindings

Type	Host Name	Port	IP Address	Binding
http		85	*	
http	testweb1	85	*	

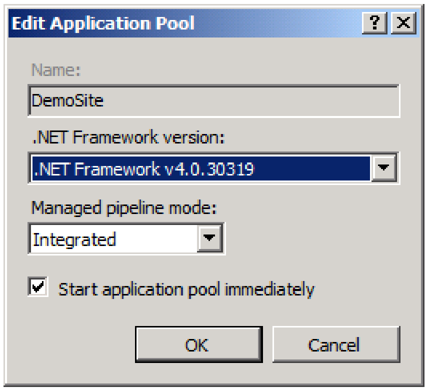
> **Note:** The first site binding allows you to access the site locally using the IP address and port or http://localhost:85. The second site binding allows you to access the site from other computers on the domain using the machine name (for example, http://testweb1:85).

13. In the **Site Bindings** dialog box, click **Close**.

14. In the **Connections** pane, click **Application Pools**.

15. In the **Application Pools** pane, right-click the name of your application pool, and then click **Basic Settings**. By default, the name of your application pool will match the name of your website (for example, **DemoSite**).

16. In the **.NET Framework version** list, select **.NET Framework v4.0.30319**, and then click **OK**.



Edit Application Pool

Name: DemoSite

.NET Framework version: .NET Framework v4.0.30319

Managed pipeline mode: Integrated

☑ Start application pool immediately

> **Note:** The sample solution requires .NET Framework 4.0. This is not a requirement for Web Deploy in general.

In order for your website to serve content, the application pool identity must have read permissions on the local folder that stores the content. In IIS 7.5, application pools run with a unique application pool identity by default (in contrast to previous versions of IIS, where application pools would typically run using the Network Service account). The application pool identity is not a real user account and does not show up on any lists of users or groups—instead, it's created dynamically when the application pool is started. Each application pool identity is added to the local **IIS_IUSRS** security group as a hidden item.

To grant permissions to an application pool identity on a file or folder, you have two options:

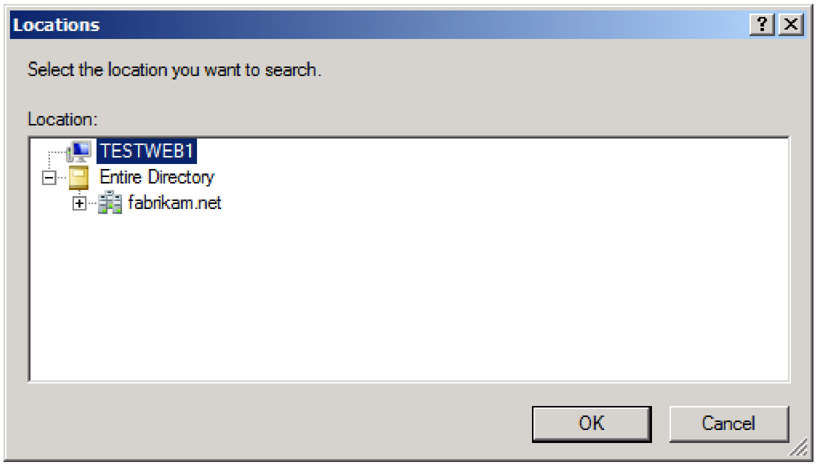
- Assign permissions to the application pool identity directly, using the format **IIS AppPool\***[application pool name]* (for example, **IIS AppPool\DemoSite**).
- Assign permissions to the **IIS_IUSRS** group.

The most common approach is to assign permissions to the local **IIS_IUSRS** group because this approach lets you change application pools without reconfiguring file system permissions. The next procedure uses this group-based approach.

Note: For more information on application pool identities in IIS 7.5, see Application Pool Identities.

To configure folder permissions for an IIS website

1. In Windows Explorer, browse to the location of your local folder.
2. Right-click the folder, and then click **Properties**.
3. On the **Security** tab, click **Edit**, and then click **Add**.
4. Click **Locations**. In the **Locations** dialog box, select the local server, and then click **OK**.



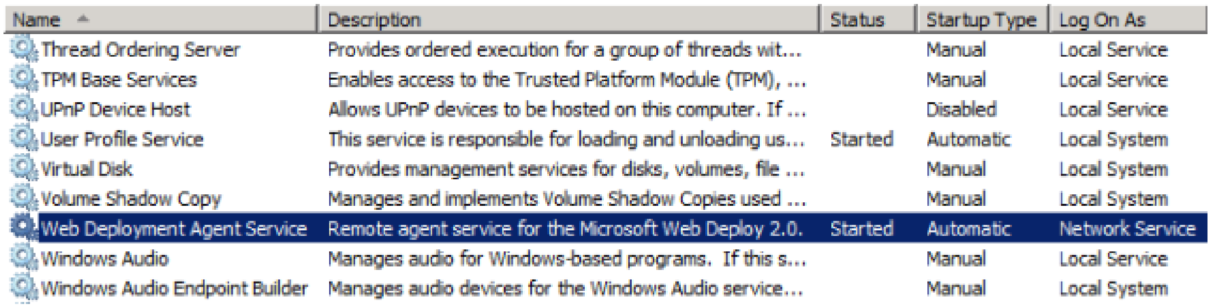
5. In the **Select Users or Groups** dialog box, type **IIS_IUSRS**, click **Check Names**, and then click **OK**.
6. In the **Permissions for** *[folder name]* dialog box, notice that the new group has been assigned the **Read & execute**, **List folder contents**, and **Read** permissions by default. Leave this unchanged and click **OK**.
7. Click **OK** to close the *[folder name]* **Properties** dialog box.

As a final task before you attempt to deploy any web packages to your server, you should ensure that the Web Deployment Agent Service is running. When you deploy a package from a remote computer, the Web Deployment Agent Service is responsible for extracting and installing the contents of the package. The service is started by default when you install the Web Deployment Tool and runs under the Network Service identity.

You can check whether a service is running in multiple different ways, using various command-line utilities or Windows PowerShell cmdlets. This procedure describes a straightforward UI-based approach.

To check that the Web Deployment Agent Service is running

1. On the **Start** menu, point to **Administrative Tools**, and then click **Services**.
2. Locate the **Web Deployment Agent Service** row, and verify that the **Status** is set to **Started**.



Name	Description	Status	Startup Type	Log On As
Thread Ordering Server	Provides ordered execution for a group of threads wit...		Manual	Local Service
TPM Base Services	Enables access to the Trusted Platform Module (TPM), ...		Manual	Local Service
UPnP Device Host	Allows UPnP devices to be hosted on this computer. If ...		Disabled	Local Service
User Profile Service	This service is responsible for loading and unloading us...	Started	Automatic	Local System
Virtual Disk	Provides management services for disks, volumes, file ...		Manual	Local System
Volume Shadow Copy	Manages and implements Volume Shadow Copies used ...		Manual	Local System
Web Deployment Agent Service	Remote agent service for the Microsoft Web Deploy 2.0.	Started	Automatic	Network Service
Windows Audio	Manages audio for Windows-based programs. If this s...		Manual	Local Service
Windows Audio Endpoint Builder	Manages audio devices for the Windows Audio service...		Manual	Local System

3. If the service is not already started, click **Start**.

Configure Firewall Exceptions

By default, the Remote Agent Service listens on TCP port 80, at this URL:

http://[*server name*]/MSDEPLOYAGENTSERVICE

In most cases, you won't need to configure any additional firewall rules for the Remote Agent Service because web servers typically listen for HTTP requests on port 80. If you customized your installation to listen on a nonstandard port, you'll need to configure firewall exceptions as required.

Conclusion

At this point, your web server is ready to accept and install web packages from a remote computer. Before you attempt to deploy a web application to the server, you may want to check these key points:

- Have you registered ASP.NET 4.0 with IIS?
- Does the application pool identity have read access to the source folder for your website?
- Is the Web Deployment Agent Service running?

Further Reading

For guidance on how to configure custom Microsoft Build Engine (MSBuild) project files to deploy web packages to the Remote Agent Service, see Configure Deployment Properties for a Target Environment.

Configuring a Web Server for Web Deploy Publishing (Web Deploy Handler)

This topic describes how to configure an Internet Information Services (IIS) web server to support web publishing and deployment using the IIS Web Deploy Handler.

When you work with Web Deploy 2.0 or later, there are three main approaches you can use to get your applications or sites onto a web server. You can:

- Use the *Web Deploy Remote Agent Service*. This approach requires less configuration of the web server, but you need to provide the credentials of a local server administrator in order to deploy anything to the server.
- Use the *Web Deploy Handler*. This approach is a lot more complex and requires more initial effort to set up the web server. However, when you use this approach, you can configure IIS to allow non-administrator users to perform the deployment. The Web Deploy Handler is only available in IIS version 7 or later.
- Use *offline deployment*. This approach requires the least configuration of the web server, but a server administrator must manually copy the web package onto the server and import it through IIS Manager.

For more information on the key features, advantages, and disadvantages of these approaches, see Choosing the Right Approach to Web Deployment.

Is the Web Deploy Handler the Right Approach for You?

Yes, if you want to allow non-administrator users to deploy content to specific IIS websites. This approach is often desirable in these types of scenarios:

- Staging or production environments, where the person or service account that triggers the remote deployment is unlikely to have access to the credentials of a server administrator.
- Hosted environments, where you want to give remote users the ability to update their websites without giving them full control of your web servers (or access to anyone else's websites).

In development or test scenarios, or in smaller organizations, deploying content using server administrator credentials is often less contentious. In these scenarios, configuring your web servers to support deployment using the Web Deploy Remote Agent Service offers a more straightforward approach.

Task Overview

To configure the web server to accept and deploy web packages from a remote computer using the Web Deploy Handler approach, you'll need to:

- Create, or choose, a domain user account (the "non-administrator user") whose credentials you'll use to perform deployments.

- Install IIS 7.5, including the Web Management Service and the Basic Authentication module.
- Install Web Deploy 2.1 or later.
- Configure the Web Management Service to allow remote connections, and start the service.
- Create an IIS website to host the deployed content.
- Grant your non-administrator user permissions on your website in IIS Manager.
- Ensure that the Web Management Service delegation rules permit the service to add and change website content using your non-administrator user account.
- Configure any firewalls to allow incoming connections on port 8172.

To host the ContactManager sample solution specifically, you'll also need to:

- Install the .NET Framework 4.0.
- Install ASP.NET MVC 3.

This topic will show you how to perform each of these procedures. The tasks and walkthroughs in this topic assume that you're starting with a clean server build running Windows Server 2008 R2. Before you continue, ensure that:

- Windows Server 2008 R2 Service Pack 1 and all available updates are installed.
- The server is domain-joined.
- The server has a static IP address.

Note: For more information on joining computers to a domain, see Joining Computers to the Domain and Logging On. For more information on configuring static IP addresses, see Configure a Static IP Address.

Install Products and Components

This section will guide you through installing the required products and components on the web server. Before you begin, a good practice is to run Windows Update to ensure that your server is fully up to date.

In this case, you need to install these things:

- **IIS 7 Recommended Configuration**. This enables the **Web Server (IIS)** role on your web server and installs the set of IIS modules and components that you need in order to host an ASP.NET application.
- **IIS: Management Service**. This installs the Web Management Service (WMSvc) in IIS. This service enables remote management of IIS websites and exposes the Web Deploy Handler endpoint to clients.

- **IIS: Basic Authentication**. This installs the IIS Basic Authentication module. This lets the Web Management Service (WMSvc) authenticate the credentials you provide.
- **Web Deployment Tool 2.1 or later**. This installs Web Deploy (and its underlying executable, MSDeploy.exe) on your server. As part of this process, it installs the Web Deploy Handler and integrates it with the Web Management Service.
- **.NET Framework 4.0**. This is required to run applications that were built on this version of the .NET Framework.
- **ASP.NET MVC 3**. This installs the assemblies you need to run MVC 3 applications.

Note: This walkthrough describes the use of the Web Platform Installer to install and configure various components. Although you don't have to use the Web Platform Installer, it simplifies the installation process by automatically detecting dependencies and ensuring that you always get the latest product versions. For more information, see Microsoft Web Platform Installer 3.0.

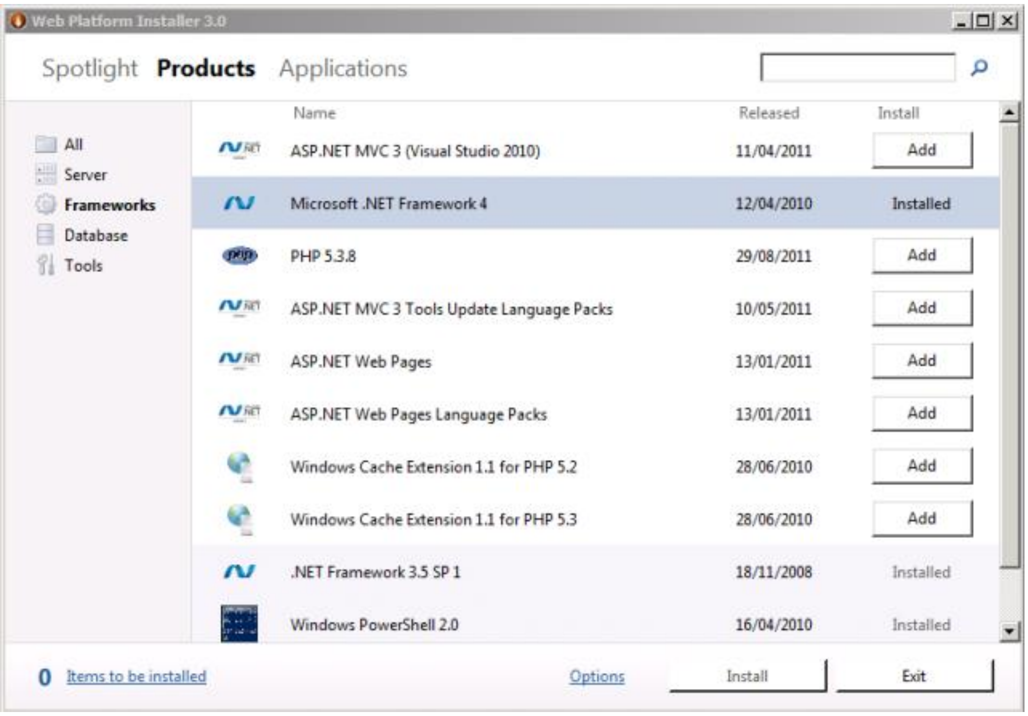
To install the required products and components

1. Download and install the Web Platform Installer.
2. When installation is complete, the Web Platform Installer will launch automatically.

   Note: You can now launch the Web Platform Installer at any time from the **Start** menu. To do this, on the **Start** menu, click **All Programs**, and then click **Microsoft Web Platform Installer**.

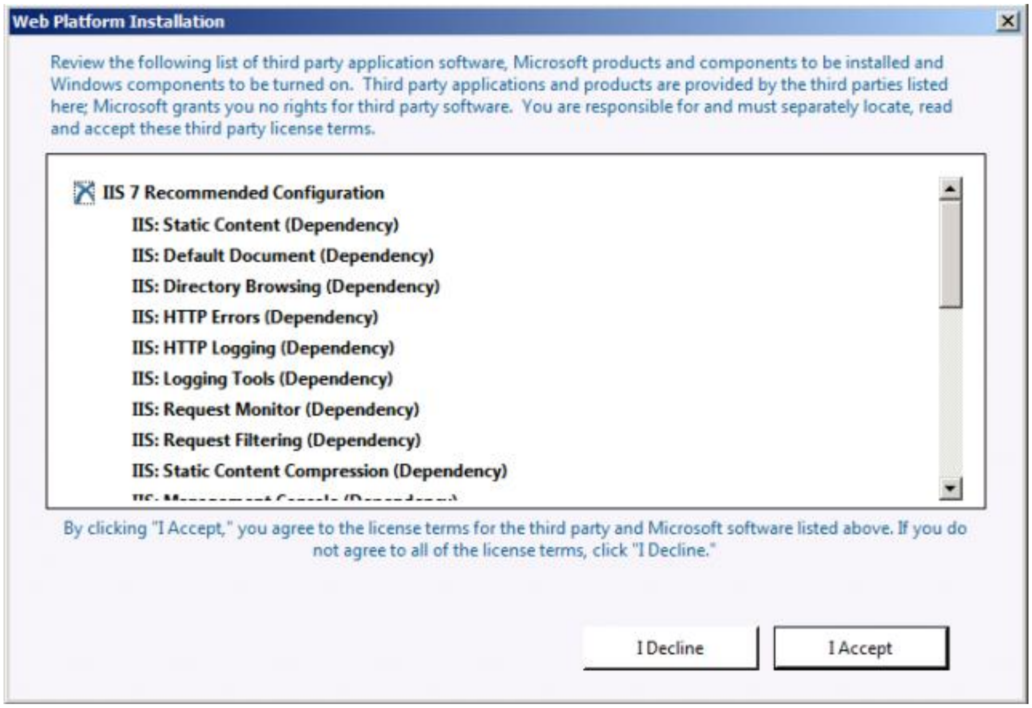
3. At the top of the **Web Platform Installer 3.0** window, click **Products**.
4. On the left side of the window, in the navigation pane, click **Frameworks**.
5. In the **Microsoft .NET Framework 4** row, if the .NET Framework is not already installed, click **Add**.

   Note: You may have already installed the .NET Framework 4.0 through Windows Update. If a product or component is already installed, the Web Platform Installer will indicate this by replacing the **Add** button with the text **Installed**.



Name	Released	Install
ASP.NET MVC 3 (Visual Studio 2010)	11/04/2011	Add
Microsoft .NET Framework 4	12/04/2010	Installed
PHP 5.3.8	29/08/2011	Add
ASP.NET MVC 3 Tools Update Language Packs	10/05/2011	Add
ASP.NET Web Pages	13/01/2011	Add
ASP.NET Web Pages Language Packs	13/01/2011	Add
Windows Cache Extension 1.1 for PHP 5.2	28/06/2010	Add
Windows Cache Extension 1.1 for PHP 5.3	28/06/2010	Add
.NET Framework 3.5 SP 1	18/11/2008	Installed
Windows PowerShell 2.0	16/04/2010	Installed

6. In the **ASP.NET MVC 3 (Visual Studio 2010)** row, click **Add**.
7. In the navigation pane, click **Server**.
8. In the **IIS 7 Recommended Configuration** row, click **Add**.
9. In the **Web Deployment Tool 2.1** row, click **Add**.
10. In the **IIS: Basic Authentication** row, click **Add**.
11. In the **IIS: Management Service** row, click **Add**.
12. Click **Install**. The Web Platform Installer will show you a list of products—together with any associated dependencies—to be installed and will prompt you to accept the license terms.



Review the following list of third party application software, Microsoft products and components to be installed and Windows components to be turned on. Third party applications and products are provided by the third parties listed here; Microsoft grants you no rights for third party software. You are responsible for and must separately locate, read and accept these third party license terms.

IIS 7 Recommended Configuration

- IIS: Static Content (Dependency)
- IIS: Default Document (Dependency)
- IIS: Directory Browsing (Dependency)
- IIS: HTTP Errors (Dependency)
- IIS: HTTP Logging (Dependency)
- IIS: Logging Tools (Dependency)
- IIS: Request Monitor (Dependency)
- IIS: Request Filtering (Dependency)
- IIS: Static Content Compression (Dependency)
- IIS: Management Console (Dependency)

By clicking "I Accept," you agree to the license terms for the third party and Microsoft software listed above. If you do not agree to all of the license terms, click "I Decline."

13. Review the license terms, and if you consent to the terms, click **I Accept**.
14. When the installation is complete, click **Finish**, and then close the **Web Platform Installer 3.0** window.

If you installed the .NET Framework 4.0 before you installed IIS, you'll need to run the ASP.NET IIS Registration Tool (aspnet_regiis.exe) to register the latest version of ASP.NET with IIS. If you don't do this, you'll find that IIS will serve static content (like HTML files) without any problems, but it will return **HTTP Error 404.0 – Not Found** when you attempt to browse to ASP.NET content. You can use the next procedure to ensure that ASP.NET 4.0 is registered.

To register ASP.NET 4.0 with IIS

1. Click **Start**, and then type **Command Prompt**.
2. In the search results, right-click **Command Prompt**, and then click **Run as administrator**.
3. In the Command Prompt window, navigate to the **%WINDIR%\Microsoft.NET\Framework\v4.0.30319** directory.
4. Type this command, and then press Enter:

```
aspnet_regiis -iru
```

5. If you plan to host 64-bit web applications at any point, you should also register the 64-bit version of ASP.NET with IIS. To do this, in the Command Prompt window, navigate to the **%WINDIR%\Microsoft.NET\Framework64\v4.0.30319** directory.
6. Type this command, and then press Enter:

```
aspnet_regiis -iru
```

As a good practice, use Windows Update again at this point to download and install any available updates for the new products and components you've installed.

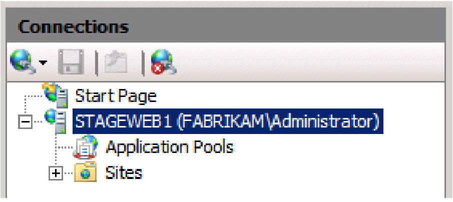
Configure the Web Management Service

Now that you've installed everything you need, the next step is to configure the Web Management Service in IIS. At a high level, you'll need to complete these tasks:

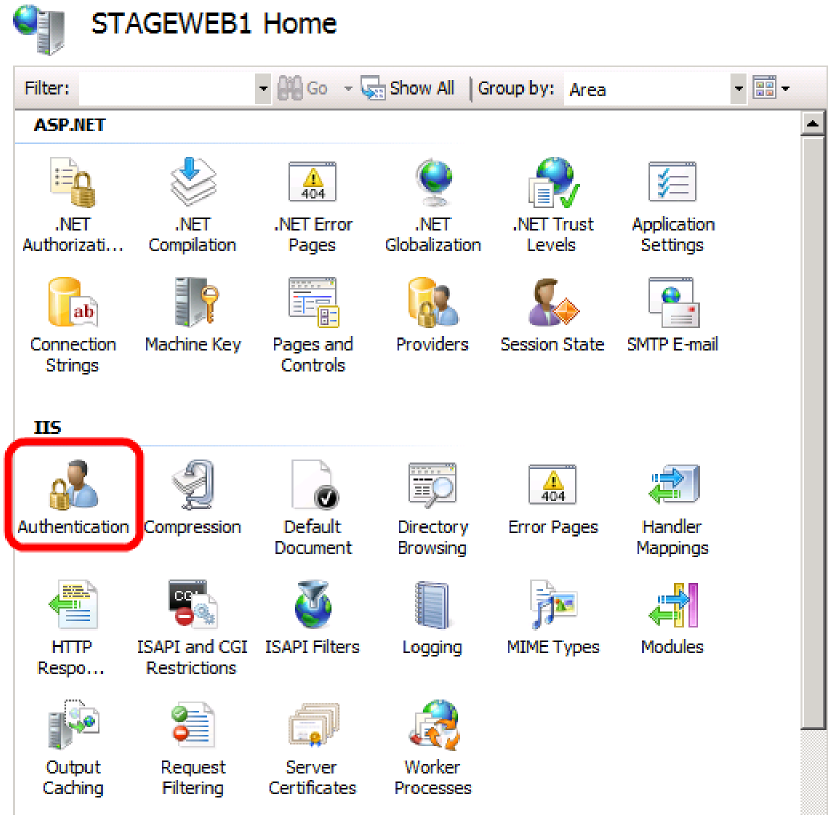
- Enable basic authentication at the server level.
- Configure the Web Management Service to accept remote connections.
- Start the Web Management Service.
- Check that the required Web Management Service delegation rules are in place.

To configure the Web Management Service

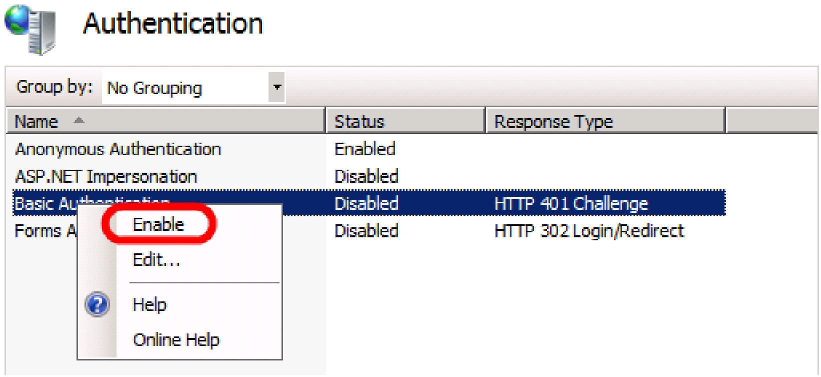
1. On the **Start** menu, point to **Administrative Tools**, and then click **Internet Information Services (IIS) Manager**.
2. In IIS Manager, in the **Connections** pane, click the server node (for example, **STAGEWEB1**).



3. In the center pane, under **IIS**, double-click **Authentication**.



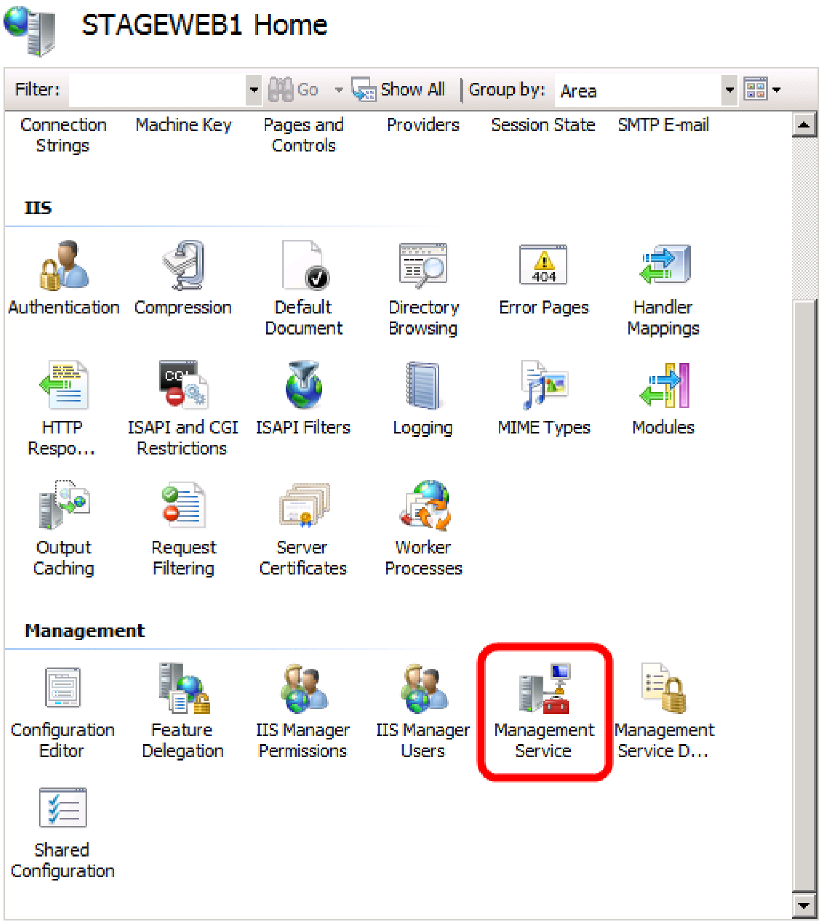
4. Right-click **Basic Authentication**, and then click **Enable**.



Name	Status	Response Type
Anonymous Authentication	Enabled	
ASP.NET Impersonation	Disabled	
Basic Au	Disabled	HTTP 401 Challenge
Forms A	Disabled	HTTP 302 Login/Redirect

5. In the **Connections** pane, click the server node again to return to the top-level settings.

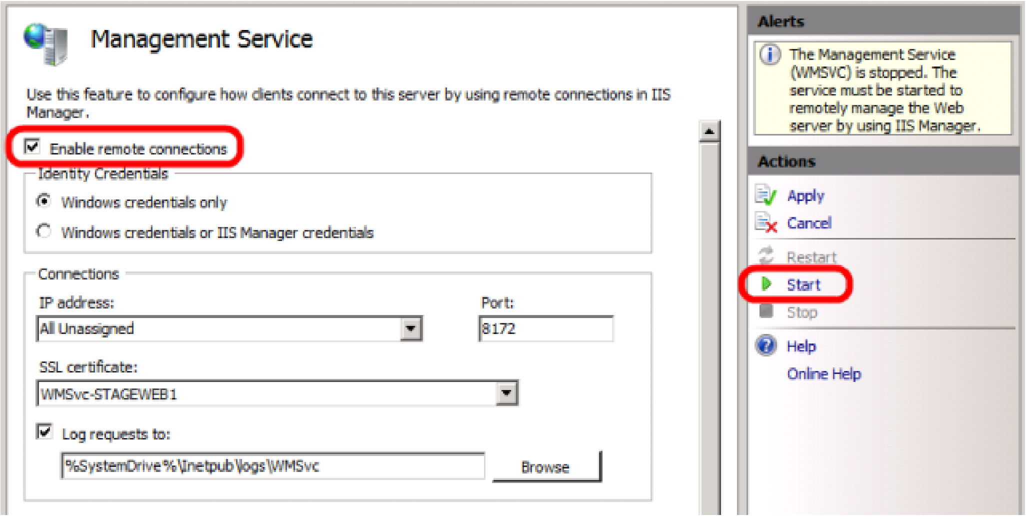
6. In the center pane, under **Management**, double-click **Management Service**.



7. In the center pane, select **Enable remote connections**.

Note: If the Web Management Service is already running, you'll need to stop it first.

8. In the **Actions** pane, click **Start** to start the Web Management Service.


Management Service

Use this feature to configure how clients connect to this server by using remote connections in IIS Manager.

☑ Enable remote connections

Identity Credentials

● Windows credentials only

○ Windows credentials or IIS Manager credentials

Connections

IP address: All Unassigned

Port: 8172

SSL certificate: WMSvc-STAGEWEB1

☑ Log requests to: %SystemDrive%\Inetpub\logs\WMSvc

Alerts

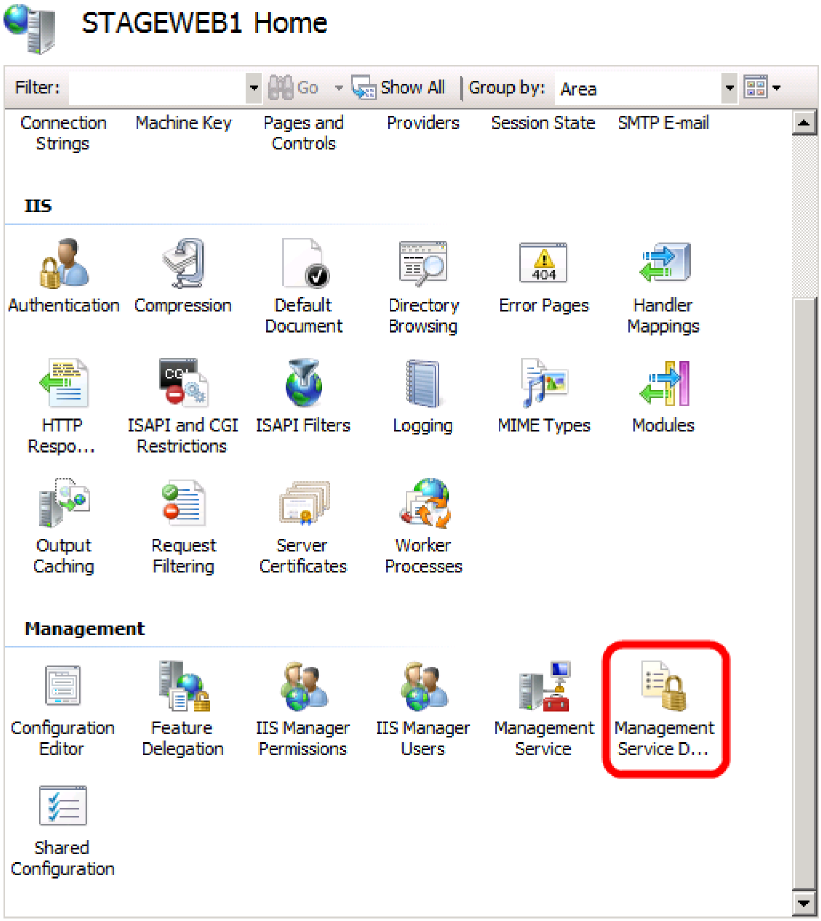
The Management Service (WMSVC) is stopped. The service must be started to remotely manage the Web server by using IIS Manager.

9. If you're prompted to save your settings, click **Yes**.

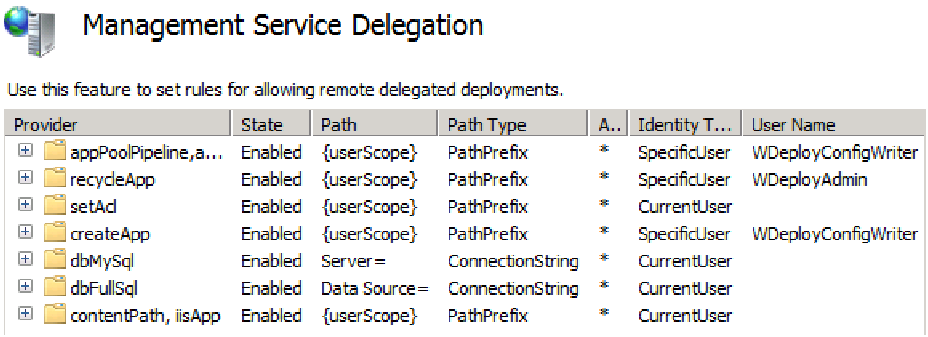
> **Note:** You may also want to configure the service to start automatically. To do this, open the Services console, right-click **Web Management Service**, and then click **Properties**. In the **Startup type** dropdown list, select **Automatic**, and then click **OK**.

10. In the **Connections** pane, click the server node again to return to the top-level settings.

11. In the center pane, under **Management**, double-click **Management Service Delegation**.



12. Verify that the center pane contains a set of rules.



Management Service Delegation

Use this feature to set rules for allowing remote delegated deployments.

Provider	State	Path	Path Type	A..	Identity T...	User Name
appPoolPipeline,a...	Enabled	{userScope}	PathPrefix	*	SpecificUser	WDeployConfigWriter
recycleApp	Enabled	{userScope}	PathPrefix	*	SpecificUser	WDeployAdmin
setAcl	Enabled	{userScope}	PathPrefix	*	CurrentUser	
createApp	Enabled	{userScope}	PathPrefix	*	SpecificUser	WDeployConfigWriter
dbMySql	Enabled	Server=	ConnectionString	*	CurrentUser	
dbFullSql	Enabled	Data Source=	ConnectionString	*	CurrentUser	
contentPath, iisApp	Enabled	{userScope}	PathPrefix	*	CurrentUser	

These rules allow authorized Web Management Service users to use various Web Deploy providers. For example, to deploy web applications and content to IIS through the Web Deploy Handler, there must be a delegation rule that allows all authenticated Web Management Service users to use the **contentPath** and **iisApp** providers (the last rule that you can see in the screenshot).

If you installed products and components in the order described in this topic, the latest version of Web Deploy should automatically add all the required delegation rules to the Web Management Service. If the Management Service Delegation page does not show any rules, you'll need to create them yourself. For instructions on how to do this, see Configure the Web Deployment Handler.

13. In the **Connections** pane, click the server node again to return to the top-level settings.

Create and Configure an IIS Website

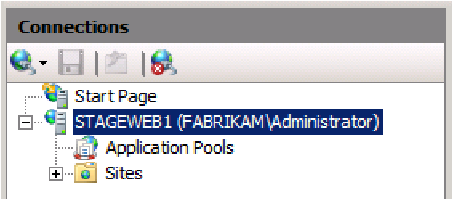
Before you can deploy web content to your server, you need to create and configure an IIS website to host the content. Web Deploy can only deploy web packages to an existing IIS website; it can't create the website for you. You also need to do a little extra configuration to allow your non-administrator account to deploy content remotely. At a high level, you'll need to complete these tasks:

- Create a folder on the file system to host your content.
- Create an IIS website to serve the content, and associate it with the local folder.
- Grant read permissions to the application pool identity on the local folder.
- Grant the necessary IIS permissions to the domain account that will deploy your web application.

Although there's nothing stopping you from deploying content to the default website in IIS, this approach is not recommended for anything other than test or demonstration scenarios. To simulate a production environment, you should create a new IIS website with settings that are specific to the requirements of your application.

To create an IIS website

1. On the local file system, create a folder to store your content (for example, **C:\DemoSite**).
2. On the **Start** menu, point to **Administrative Tools**, and then click **Internet Information Services (IIS) Manager**.
3. In IIS Manager, in the **Connections** pane, expand the server node (for example, **STAGEWEB1**).



4. Right-click the **Sites** node, and then click **Add Web Site**.

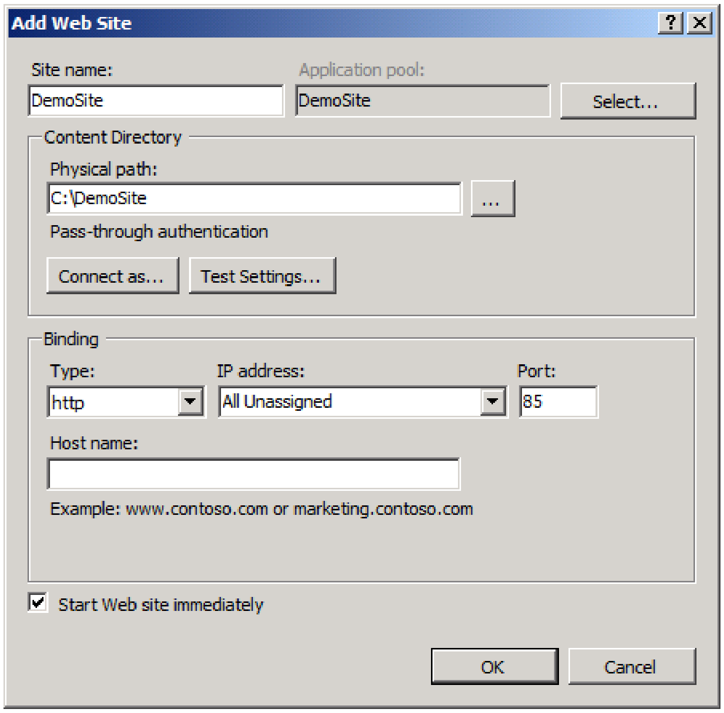
5. In the **Site name** box, type a name for the IIS website (for example, **DemoSite**).

6. In the **Physical path** box, type (or browse to) the path to your local folder (for example, **C:\DemoSite**).

7. In the **Port** box, type the port number on which you want to host the website (for example, **85**).

> **Note:** The standard port numbers are 80 for HTTP and 443 for HTTPS. However, if you host this website on port 80, you'll need to stop the default website before you can access your site.

8. Leave the **Host name** box blank, unless you want to configure a Domain Name System (DNS) record for the website, and then click **OK**.



Site name: DemoSite

Application pool: DemoSite

Physical path: C:\DemoSite

Type: http

IP address: All Unassigned

Port: 85

Host name:

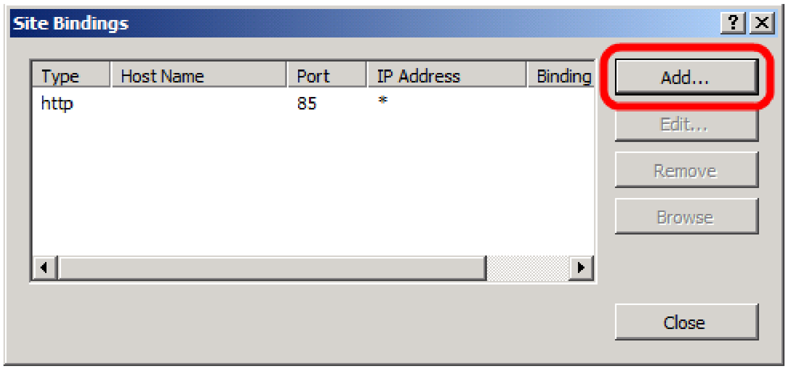
Example: www.contoso.com or marketing.contoso.com

☑ Start Web site immediately

> **Note:** In a production environment, you'll likely want to host your website on port 80 and configure a host header, together with matching DNS records. For more information on configuring host headers in IIS 7, see Configure a Host Header for a Web Site (IIS 7). For more information on the DNS Server role in Windows Server 2008 R2, see DNS Server Overview and DNS Server.

9. In the **Actions** pane, under **Edit Site**, click **Bindings**.

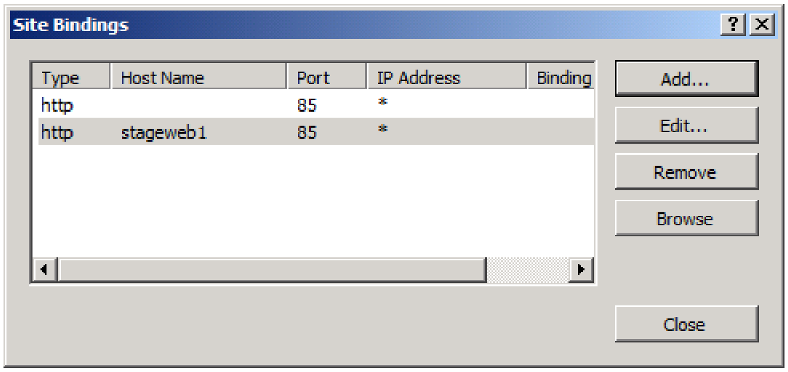
10. In the **Site Bindings** dialog box, click **Add**.



Type	Host Name	Port	IP Address	Binding
http		85	*	

11. In the **Add Site Binding** dialog box, set the **IP address** and **Port** to match your existing site configuration.

12. In the **Host name** box, type the name of your web server (for example, **STAGEWEB1**), and then click **OK**.



Type	Host Name	Port	IP Address	Binding
http		85	*	
http	stageweb1	85	*	

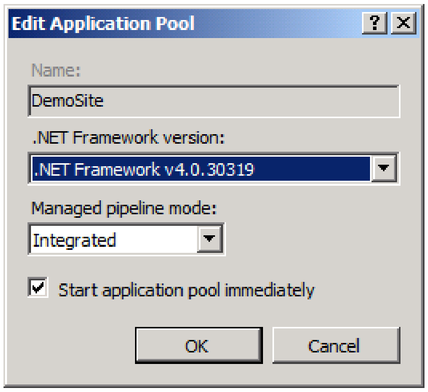
> **Note:** The first site binding allows you to access the site locally using the IP address and port or http://localhost:85. The second site binding allows you to access the site from other computers on the domain using the machine name (for example, http://stageweb1:85).

13. In the **Site Bindings** dialog box, click **Close**.

14. In the **Connections** pane, click **Application Pools**.

15. In the **Application Pools** pane, right-click the name of your application pool, and then click **Basic Settings**. By default, the name of your application pool will match the name of your website (for example, **DemoSite**).

16. In the **.NET Framework version** list, select **.NET Framework v4.0.30319**, and then click **OK**.



DemoSite

.NET Framework version: .NET Framework v4.0.30319

Managed pipeline mode: Integrated

☑ Start application pool immediately

Note: The sample solution requires .NET Framework 4.0. This is not a requirement for Web Deploy in general.

In order for your website to serve content, the application pool identity must have read permissions on the local folder that stores the content. In IIS 7.5, application pools run with a unique application pool identity by default (in contrast to previous versions of IIS, where application pools would typically run using the Network Service account). The application pool identity is not a real user account and does not show up on any lists of users or groups—instead, it's created dynamically when the application pool is started. Each application pool identity is added to the local **IIS_IUSRS** security group as a hidden item.

To grant permissions to an application pool identity on a file or folder, you have two options:

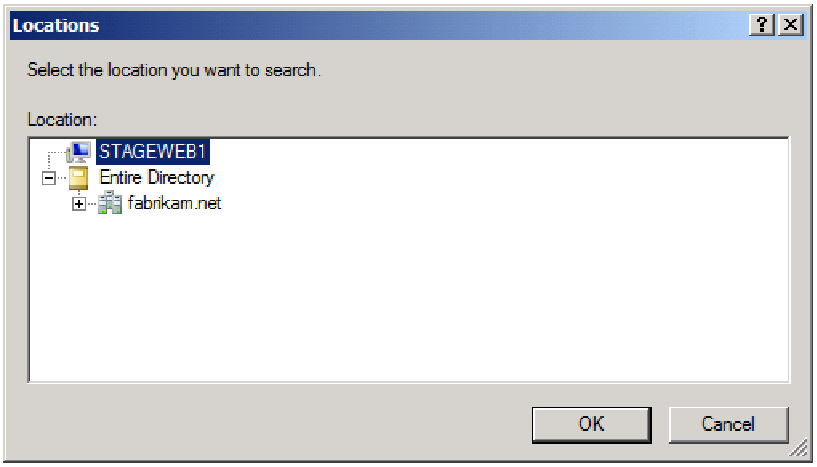
- Assign permissions to the application pool identity directly, using the format **IIS AppPool\***[application pool name]* (for example, **IIS AppPool\DemoSite**).
- Assign permissions to the **IIS_IUSRS** group.

The most common approach is to assign permissions to the local **IIS_IUSRS** group, because this approach lets you change application pools without reconfiguring file system permissions. The next procedure uses this group-based approach.

Note: For more information on application pool identities in IIS 7.5, see Application Pool Identities.

To configure folder permissions for an IIS website

1. In Windows Explorer, browse to the location of your local folder.
2. Right-click the folder, and then click **Properties**.
3. On the **Security** tab, click **Edit**, and then click **Add**.
4. Click **Locations**. In the **Locations** dialog box, select the local server, and then click **OK**.

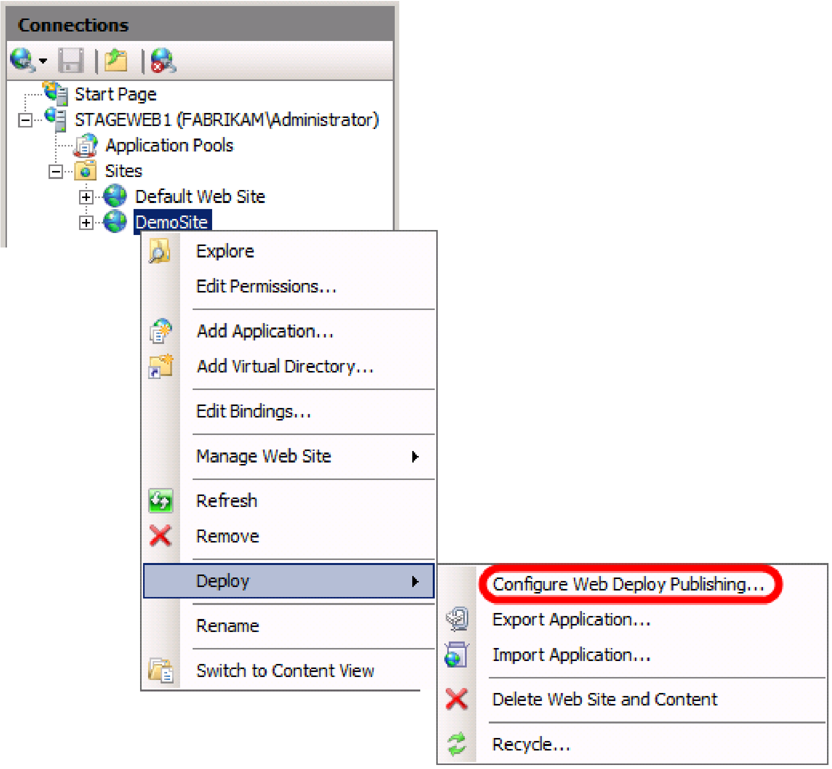


5. In the **Select Users or Groups** dialog box, type **IIS_IUSRS**, click **Check Names**, and then click **OK**.
6. In the **Permissions for** *[folder name]* dialog box, notice that the new group has been assigned the **Read & execute**, **List folder contents**, and **Read** permissions by default. Leave this unchanged and click **OK**.
7. Click **OK** to close the *[folder name]* **Properties** dialog box.

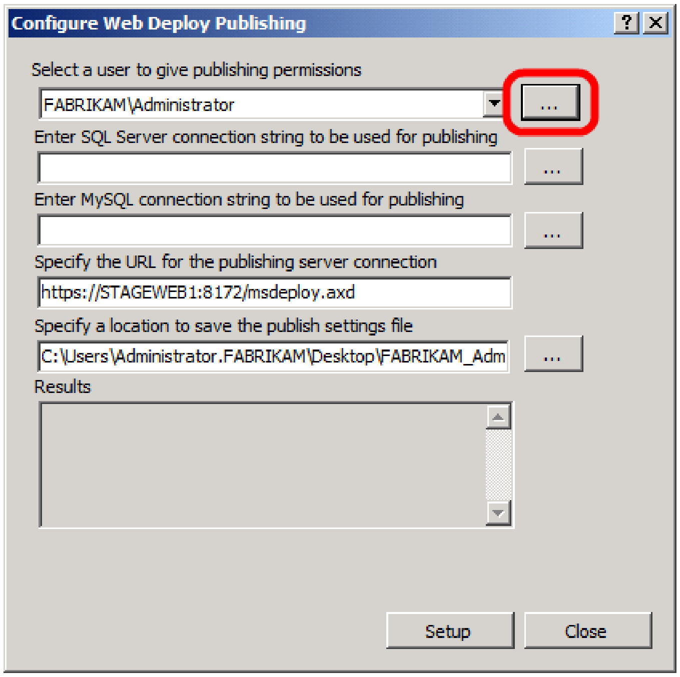
As a final task, you must grant the appropriate permissions to the non-administrator user whose credentials you'll use to deploy content. This user requires the permissions to deploy content remotely to your website.

To configure IIS website permissions for a non-administrator domain user

1. In IIS Manager, in the **Connections** pane, right-click your website node (for example, **DemoSite**), point to **Deploy**, and then click **Configure Web Deploy Publishing**.



2. In the **Configure Web Deploy Publishing** dialog box, to the right of the **Select a user to give publishing permissions** list, click the ellipsis button.



Configure Web Deploy Publishing

Select a user to give publishing permissions: FABRIKAM\Administrator

Enter SQL Server connection string to be used for publishing:

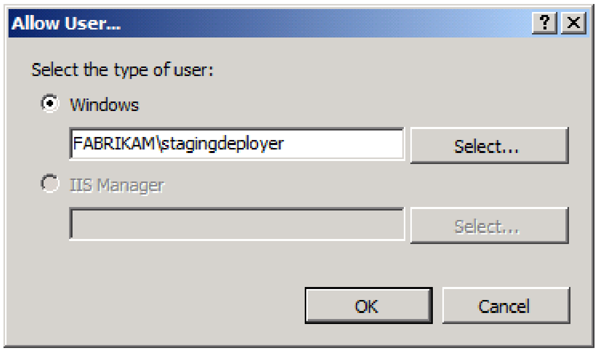
Enter MySQL connection string to be used for publishing:

Specify the URL for the publishing server connection: https://STAGEWEB1:8172/msdeploy.axd

Specify a location to save the publish settings file: C:\Users\Administrator.FABRIKAM\Desktop\FABRIKAM_Adm

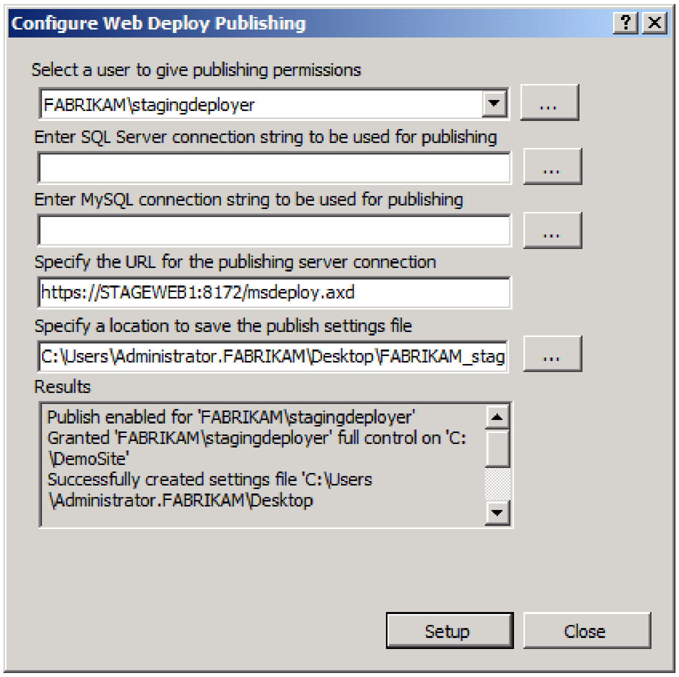
Results

3. In the **Allow User** dialog box, type the domain and user name of the account you want to use to deploy content, and then click **OK**.



FABRIKAM\stagingdeployer

4. In the **Configure Web Deploy Publishing** dialog box, click **Setup**.



Select a user to give publishing permissions: FABRIKAM\stagingdeployer

Enter SQL Server connection string to be used for publishing:

Enter MySQL connection string to be used for publishing:

Specify the URL for the publishing server connection: https://STAGEWEB1:8172/msdeploy.axd

Specify a location to save the publish settings file: C:\Users\Administrator.FABRIKAM\Desktop\FABRIKAM_stag

Results

```
Publish enabled for 'FABRIKAM\stagingdeployer'
Granted 'FABRIKAM\stagingdeployer' full control on 'C:
\DemoSite'
Successfully created settings file 'C:\Users
\Administrator.FABRIKAM\Desktop
```

Note: This operation performs two key functions in one step. First, it grants the user permission to modify the website remotely through the Web Management Service, according to the delegation rules you examined in the previous section. Second, it grants the user full control of the source folder for the website, which allows the user to add, modify, and set permissions on the website content.

5. In the **Configure Web Deploy Publishing** dialog box, click **Close**.

Configure Firewall Exceptions

By default, the IIS Web Management Service listens on TCP port 8172. If Windows Firewall is enabled on your web server, you'll need to create a new inbound rule to allow TCP traffic on port 8172 (all outbound traffic is permitted by default in Windows Firewall). If you use a third-party firewall, you'll need to create rules to allow traffic.

Direction	From Port	To Port	Port Type
Inbound	Any	8172	TCP
Outbound	8172	Any	TCP

For more information on configuring rules in Windows Firewall, see Configuring Firewall Rules. For third-party firewalls, please consult your product documentation.

Conclusion

Your web server should now be ready to accept remote deployments to the Web Deploy Handler through the Web Management Service. Before you attempt to deploy a web application to the server, you may want to check these key points:

- Have you enabled basic authentication at the server level in IIS?
- Have you enabled remote connections to the Web Management Service?
- Have you started the Web Management Service?
- Are there management service delegation rules in place?
- Does the application pool identity have read access to the source folder for your website?
- Does the non-administrator user account have site-level permissions in IIS?
- Does your firewall allow incoming connections to the server on TCP port 8172?

Further Reading

For guidance on how to configure custom Microsoft Build Engine (MSBuild) project files to deploy web packages to the Web Deploy Handler, see Configure Deployment Properties for a Target Environment.

Configuring a Web Server for Web Deploy Publishing (Offline Deployment)

This topic describes how to configure an IIS web server to support offline web publishing and deployment.

When you work with Internet Information Services (IIS) Web Deployment Tool (Web Deploy) 2.0 or later, there are three main approaches you can use to get your applications or sites onto a web server. You can:

- Use the *Web Deploy Remote Agent Service*. This approach requires less configuration of the web server, but you need to provide the credentials of a local server administrator in order to deploy anything to the server.
- Use the *Web Deploy Handler*. This approach is a lot more complex and requires more initial effort to set up the web server. However, when you use this approach, you can configure IIS to

allow non-administrator users to perform the deployment. The Web Deploy Handler is only available in IIS version 7 or later.

- Use *offline deployment*. This approach requires the least configuration of the web server, but a server administrator must manually copy the web package onto the server and import it through IIS Manager.

For more information on the key features, advantages, and disadvantages of these approaches, see Choosing the Right Approach to Web Deployment.

Is Offline Deployment the Right Approach for You?

Yes, if your network infrastructure or security restrictions prevent remote deployment. This is most likely to be the case in Internet-facing production environments, where the web servers are isolated—either physically or by firewalls and subnets—from the rest of your server infrastructure.

Obviously, this approach becomes less desirable if your web applications are updated on a regular basis. If your infrastructure allows it, you may want to consider enabling remote deployment, using either the Web Deploy Handler or the Web Deploy Remote Agent Service.

Task Overview

To configure the web server to support offline import and deployment of web packages, you'll need to:

- Install IIS 7.5 and the IIS 7 recommended configuration.
- Install Web Deploy 2.1 or later.
- Create an IIS website to host the deployed content.
- Disable the Web Deployment Agent Service.

To host the sample solution specifically, you'll also need to:

- Install the .NET Framework 4.0.
- Install ASP.NET MVC 3.

This topic will show you how to perform each of these procedures. The tasks and walkthroughs in this topic assume that you're starting with a clean server build running Windows Server 2008 R2. Before you continue, ensure that:

- Windows Server 2008 R2 Service Pack 1 and all available updates are installed.
- The server is domain-joined.
- The server has a static IP address.

Note: For more information on joining computers to a domain, see Joining Computers to the Domain and Logging On. For more information on configuring static IP addresses, see Configure a Static IP Address.

Install Products and Components

This section will guide you through installing the required products and components on the web server. Before you begin, a good practice is to run Windows Update to ensure that your server is fully up to date.

In this case, you need to install these things:

- **IIS 7 Recommended Configuration**. This enables the **Web Server (IIS)** role on your web server and installs the set of IIS modules and components that you need in order to host an ASP.NET application.
- **.NET Framework 4.0**. This is required to run applications that were built on this version of the .NET Framework.
- **Web Deployment Tool 2.1 or later**. This installs Web Deploy (and its underlying executable, MSDeploy.exe) on your server. Web Deploy integrates with IIS and lets you import and export web packages.
- **ASP.NET MVC 3**. This installs the assemblies you need to run MVC 3 applications.

Note: This walkthrough describes the use of the Web Platform Installer to install and configure various components. Although you don't have to use the Web Platform Installer, it simplifies the installation process by automatically detecting dependencies and ensuring that you always get the latest product versions. For more information, see Microsoft Web Platform Installer 3.0.

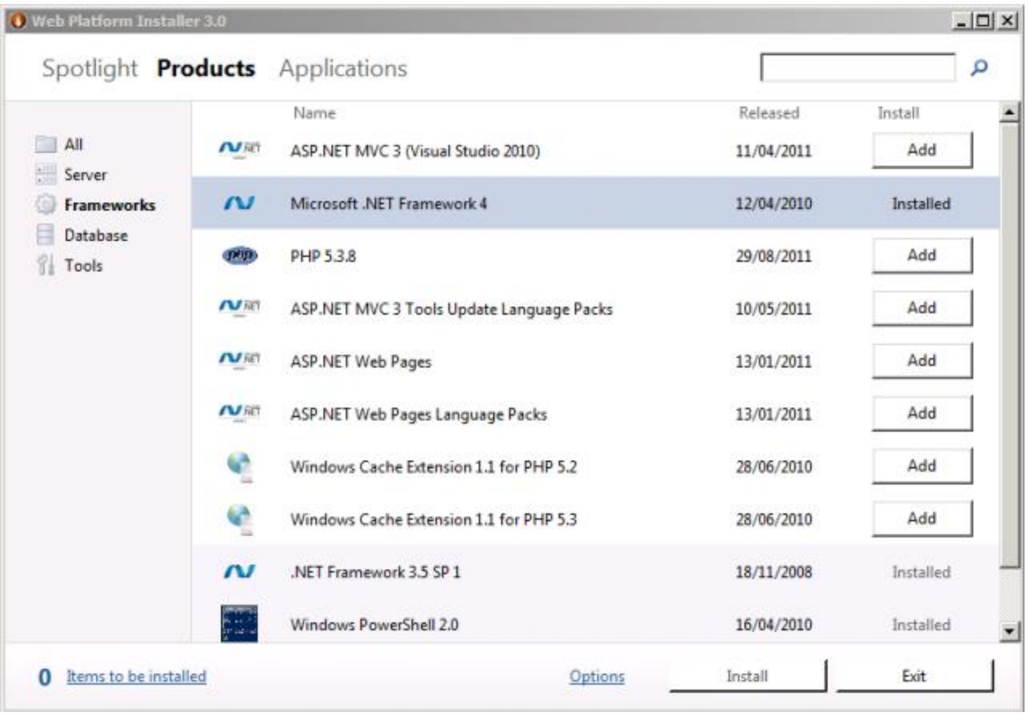
To install the required products and components

1. Download and install the Web Platform Installer.
2. When installation is complete, the Web Platform Installer will launch automatically.

   Note: You can now launch the Web Platform Installer at any time from the **Start** menu. To do this, on the **Start** menu, click **All Programs**, and then click **Microsoft Web Platform Installer**.

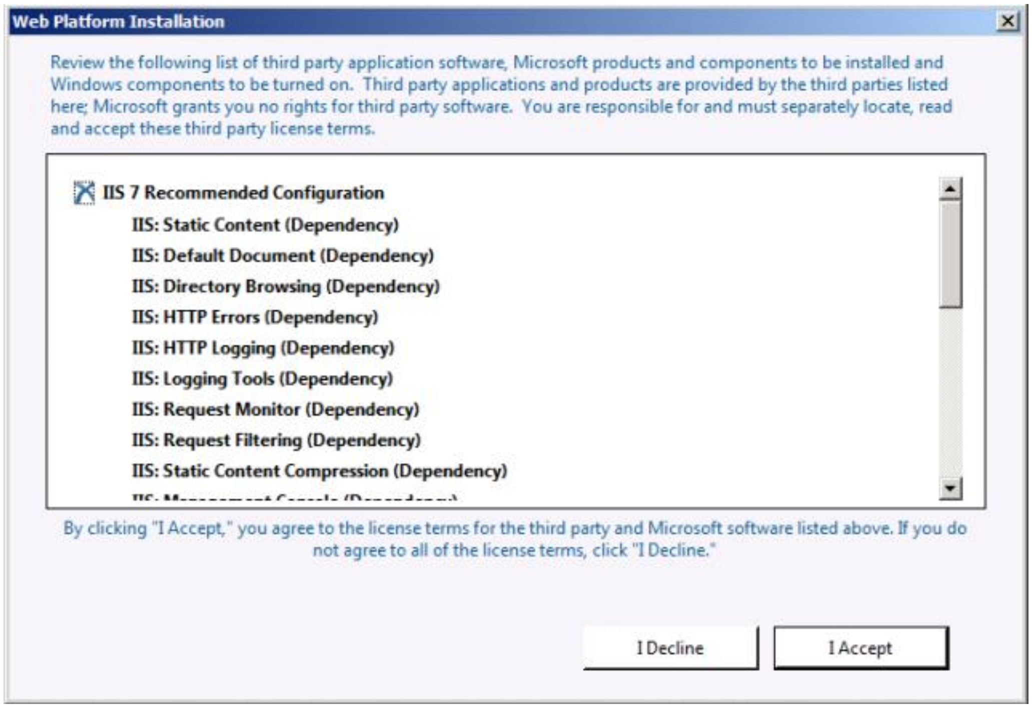
3. At the top of the **Web Platform Installer 3.0** window, click **Products**.
4. On the left side of the window, in the navigation pane, click **Frameworks**.
5. In the **Microsoft .NET Framework 4** row, if the .NET Framework is not already installed, click **Add**.

   Note: You may have already installed the .NET Framework 4.0 through Windows Update. If a product or component is already installed, the Web Platform Installer will indicate this by replacing the **Add** button with the text **Installed**.



Name	Released	Install
ASP.NET MVC 3 (Visual Studio 2010)	11/04/2011	Add
Microsoft .NET Framework 4	12/04/2010	Installed
PHP 5.3.8	29/08/2011	Add
ASP.NET MVC 3 Tools Update Language Packs	10/05/2011	Add
ASP.NET Web Pages	13/01/2011	Add
ASP.NET Web Pages Language Packs	13/01/2011	Add
Windows Cache Extension 1.1 for PHP 5.2	28/06/2010	Add
Windows Cache Extension 1.1 for PHP 5.3	28/06/2010	Add
.NET Framework 3.5 SP 1	18/11/2008	Installed
Windows PowerShell 2.0	16/04/2010	Installed

6. In the **ASP.NET MVC 3 (Visual Studio 2010)** row, click **Add**.
7. In the navigation pane, click **Server**.
8. In the **IIS 7 Recommended Configuration** row, click **Add**.
9. In the **Web Deployment Tool 2.1** row, click **Add**.
10. Click **Install**. The Web Platform Installer will show you a list of products—together with any associated dependencies—to be installed and will prompt you to accept the license terms.



Web Platform Installation

Review the following list of third party application software, Microsoft products and components to be installed and Windows components to be turned on. Third party applications and products are provided by the third parties listed here; Microsoft grants you no rights for third party software. You are responsible for and must separately locate, read and accept these third party license terms.

IIS 7 Recommended Configuration
IIS: Static Content (Dependency)
IIS: Default Document (Dependency)
IIS: Directory Browsing (Dependency)
IIS: HTTP Errors (Dependency)
IIS: HTTP Logging (Dependency)
IIS: Logging Tools (Dependency)
IIS: Request Monitor (Dependency)
IIS: Request Filtering (Dependency)
IIS: Static Content Compression (Dependency)

By clicking "I Accept," you agree to the license terms for the third party and Microsoft software listed above. If you do not agree to all of the license terms, click "I Decline."

11. Review the license terms, and if you consent to the terms, click **I Accept**.
12. When the installation is complete, click **Finish**, and then close the **Web Platform Installer 3.0** window.

If you installed the .NET Framework 4.0 before you installed IIS, you'll need to run the ASP.NET IIS Registration Tool (aspnet_regiis.exe) to register the latest version of ASP.NET with IIS. If you don't do this, you'll find that IIS will serve static content (like HTML files) without any problems, but it will return **HTTP Error 404.0 – Not Found** when you attempt to browse to ASP.NET content. You can use the next procedure to ensure that ASP.NET 4.0 is registered.

To register ASP.NET 4.0 with IIS

1. Click **Start**, and then type **Command Prompt**.
2. In the search results, right-click **Command Prompt**, and then click **Run as administrator**.
3. In the Command Prompt window, navigate to the **%WINDIR%\Microsoft.NET\Framework\v4.0.30319** directory.
4. Type this command, and then press Enter:

```
aspnet_regiis -iru
```

5. If you plan to host 64-bit web applications at any point, you should also register the 64-bit version of ASP.NET with IIS. To do this, in the Command Prompt window, navigate to the **%WINDIR%\Microsoft.NET\Framework64\v4.0.30319** directory.
6. Type this command, and then press Enter:

```
aspnet_regiis -iru
```

As a good practice, use Windows Update again at this point to download and install any available updates for the new products and components you've installed.

Configure the IIS Website

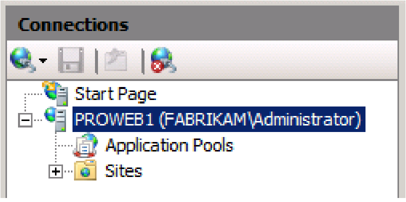
Before you can deploy web content to your server, you need to create and configure an IIS website to host the content. Web Deploy can only deploy web packages to an existing IIS website; it can't create the website for you. At a high level, you'll need to complete these tasks:

- Create a folder on the file system to host your content.
- Create an IIS website to serve the content, and associate it with the local folder.
- Grant read permissions to the application pool identity on the local folder.

Although there's nothing stopping you from deploying content to the default website in IIS, this approach is not recommended for anything other than test or demonstration scenarios. To simulate a production environment, you should create a new IIS website with settings that are specific to the requirements of your application.

To create and configure an IIS website

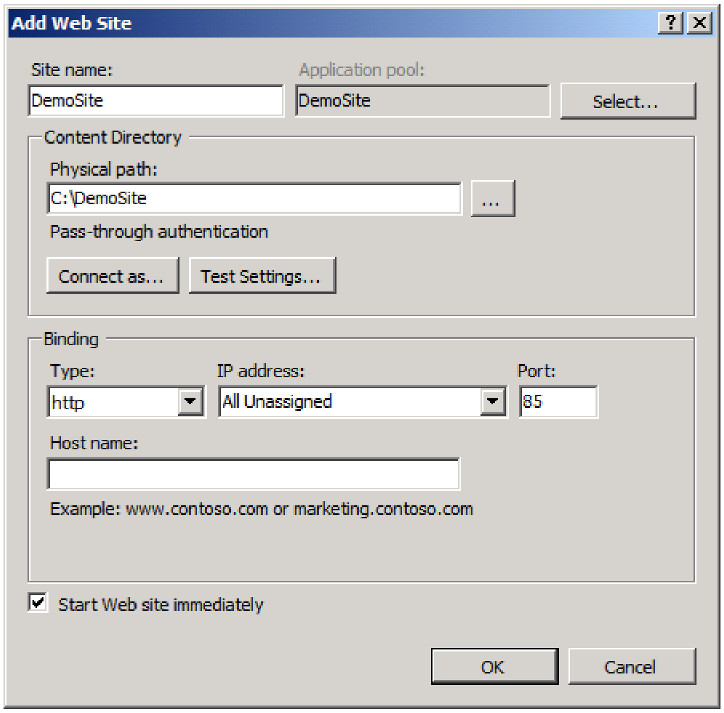
1. On the local file system, create a folder to store your content (for example, **C:\DemoSite**).
2. On the **Start** menu, point to **Administrative Tools**, and then click **Internet Information Services (IIS) Manager**.
3. In IIS Manager, in the **Connections** pane, expand the server node (for example, **PROWEB1**).



4. Right-click the **Sites** node, and then click **Add Web Site**.
5. In the **Site name** box, type a name for the IIS website (for example, **DemoSite**).
6. In the **Physical path** box, type (or browse to) the path to your local folder (for example, **C:\DemoSite**).
7. In the **Port** box, type the port number on which you want to host the website (for example, **85**).

> **Note:** The standard port numbers are 80 for HTTP and 443 for HTTPS. However, if you host this website on port 80, you'll need to stop the default website before you can access your site.

8. Leave the **Host name** box blank, unless you want to configure a Domain Name System (DNS) record for the website, and then click **OK**.



Site name: DemoSite

Application pool: DemoSite

Physical path: C:\DemoSite

Type: http

IP address: All Unassigned

Port: 85

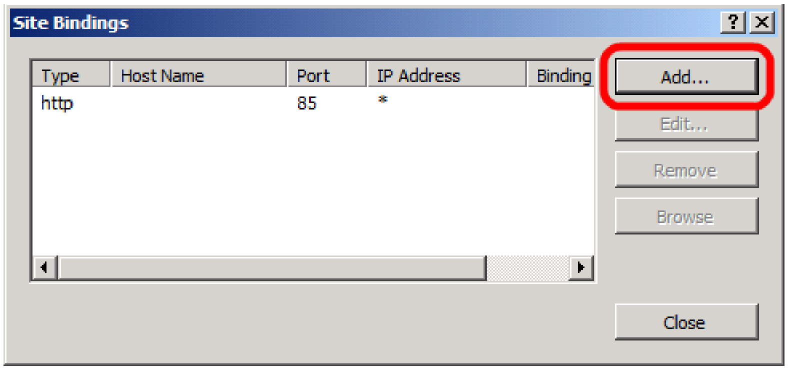
Host name:

☑ Start Web site immediately

Note: In a production environment, you'll likely want to host your website on port 80 and configure a host header, together with matching DNS records. For more information on configuring host headers in IIS 7, see Configure a Host Header for a Web Site (IIS 7). For more information on the DNS Server role in Windows Server 2008 R2, see DNS Server Overview and DNS Server.

9. In the **Actions** pane, under **Edit Site**, click **Bindings**.

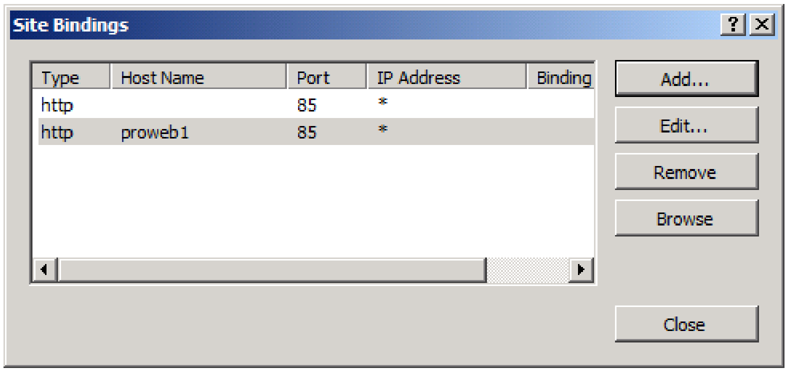
10. In the **Site Bindings** dialog box, click **Add**.



Type	Host Name	Port	IP Address	Binding
http		85	*	

11. In the **Add Site Binding** dialog box, set the **IP address** and **Port** to match your existing site configuration.

12. In the **Host name** box, type the name of your web server (for example, **PROWEB1**), and then click **OK**.



Site Bindings

Type	Host Name	Port	IP Address	Binding
http		85	*	
http	proweb1	85	*	

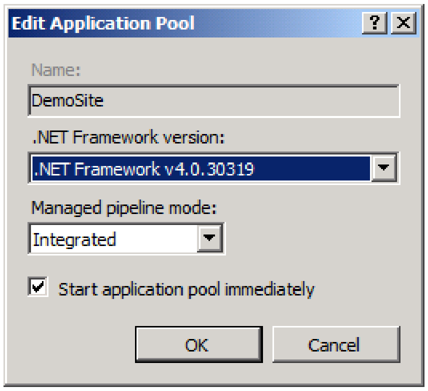
Note: The first site binding allows you to access the site locally using the IP address and port or http://localhost:85. The second site binding allows you to access the site from other computers on the domain using the machine name (for example, http://proweb1:85).

13. In the **Site Bindings** dialog box, click **Close**.

14. In the **Connections** pane, click **Application Pools**.

15. In the **Application Pools** pane, right-click the name of your application pool, and then click **Basic Settings**. By default, the name of your application pool will match the name of your website (for example, **DemoSite**).

16. In the **.NET Framework version** list, select **.NET Framework v4.0.30319**, and then click **OK**.



Edit Application Pool

Name:
DemoSite

.NET Framework version:
.NET Framework v4.0.30319

Managed pipeline mode:
Integrated

☑ Start application pool immediately

Note: The sample solution requires .NET Framework 4.0. This is not a requirement for Web Deploy in general.

In order for your website to serve content, the application pool identity must have read permissions on the local folder that stores the content. In IIS 7.5, application pools run with a unique application pool identity by default (in contrast to previous versions of IIS, where application pools would typically run using the Network Service account). The application pool identity is not a real user account and does not show up on any lists of users or groups—instead, it's created dynamically when the application pool is started. Each application pool identity is added to the local **IIS_IUSRS** security group as a hidden item.

To grant permissions to an application pool identity on a file or folder, you have two options:

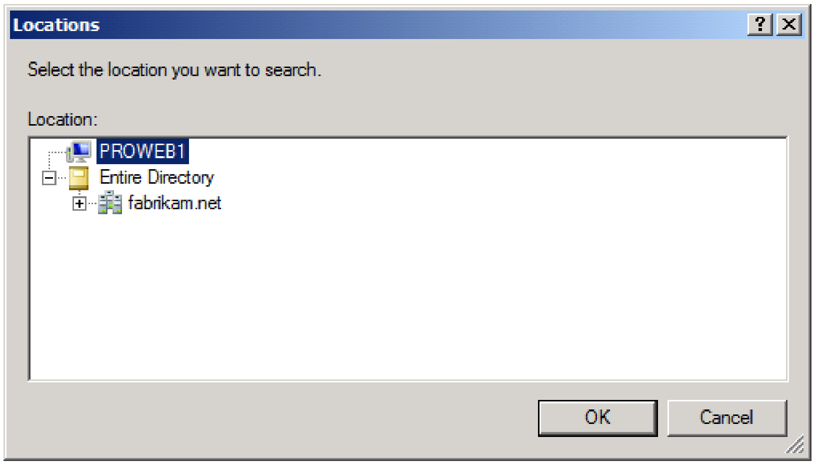
- Assign permissions to the application pool identity directly, using the format **IIS AppPool\***[application pool name]* (for example, **IIS AppPool\DemoSite**).
- Assign permissions to the **IIS_IUSRS** group.

The most common approach is to assign permissions to the local **IIS_IUSRS** group, because this approach lets you change application pools without reconfiguring file system permissions. The next procedure uses this group-based approach.

Note: For more information on application pool identities in IIS 7.5, see Application Pool Identities.

To configure folder permissions for an IIS website

1. In Windows Explorer, browse to the location of your local folder.
2. Right-click the folder, and then click **Properties**.
3. On the **Security** tab, click **Edit**, and then click **Add**.
4. Click **Locations**. In the **Locations** dialog box, select the local server, and then click **OK**.



5. In the **Select Users or Groups** dialog box, type **IIS_IUSRS**, click **Check Names**, and then click **OK**.
6. In the **Permissions for** *[folder name]* dialog box, notice that the new group has been assigned the **Read & execute**, **List folder contents**, and **Read** permissions by default. Leave this unchanged and click **OK**.
7. Click **OK** to close the *[folder name]* **Properties** dialog box.

Disable the Remote Agent Service

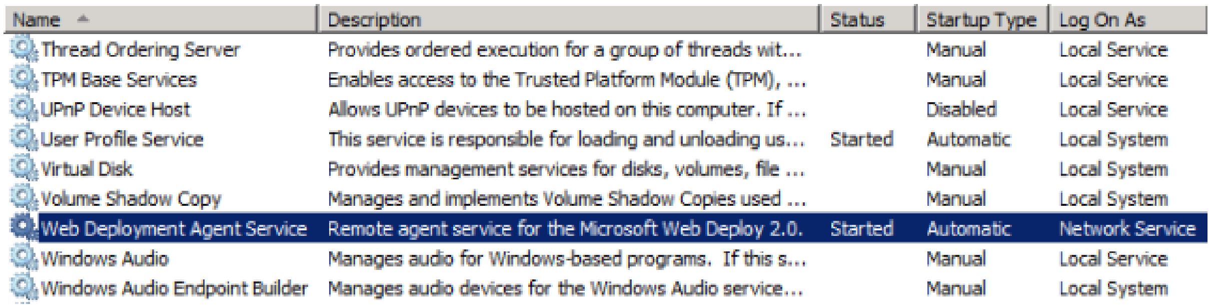
When you install Web Deploy, the Web Deployment Agent Service is installed and started automatically. This service allows you to deploy and publish web packages from a remote location. You won't be using the remote deployment capability in this scenario, so you should stop and disable the service.

> **Note:** You don't need to stop the remote agent service in order to import and deploy a web package manually. However, it's a good practice to stop and disable the service if you don't plan to use it.

You can stop and disable a service in multiple ways, using various command-line utilities or Windows PowerShell cmdlets. This procedure describes a straightforward UI-based approach.

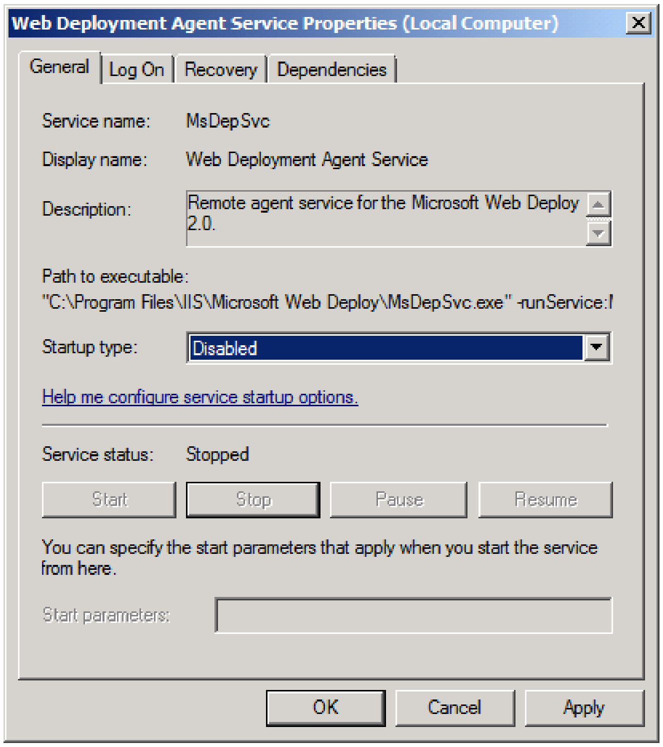
To stop and disable the remote agent service

1. On the **Start** menu, point to **Administrative Tools**, and then click **Services**.
2. In the Services console, locate the **Web Deployment Agent Service** row.



Name	Description	Status	Startup Type	Log On As
Thread Ordering Server	Provides ordered execution for a group of threads wit...		Manual	Local Service
TPM Base Services	Enables access to the Trusted Platform Module (TPM), ...		Manual	Local Service
UPnP Device Host	Allows UPnP devices to be hosted on this computer. If ...		Disabled	Local Service
User Profile Service	This service is responsible for loading and unloading us...	Started	Automatic	Local System
Virtual Disk	Provides management services for disks, volumes, file ...		Manual	Local System
Volume Shadow Copy	Manages and implements Volume Shadow Copies used ...		Manual	Local System
Web Deployment Agent Service	Remote agent service for the Microsoft Web Deploy 2.0.	Started	Automatic	Network Service
Windows Audio	Manages audio for Windows-based programs. If this s...		Manual	Local Service
Windows Audio Endpoint Builder	Manages audio devices for the Windows Audio service...		Manual	Local System

3. Right-click **Web Deployment Agent Service**, and then click **Properties**.
4. In the **Web Deployment Agent Service Properties** dialog box, click **Stop**.
5. In the **Startup type** list, select **Disabled**, and then click **OK**.



Service name: MsDepSvc

Display name: Web Deployment Agent Service

Description: Remote agent service for the Microsoft Web Deploy 2.0.

Path to executable:
"C:\Program Files\IIS\Microsoft Web Deploy\MsDepSvc.exe" -runService:

Startup type: Disabled

Help me configure service startup options.

Service status: Stopped

You can specify the start parameters that apply when you start the service from here.

Start parameters:

Conclusion

At this point, your web server is ready for offline web package deployment. Before you attempt to import web packages to an IIS website, you may want to check these key points:

- Have you registered ASP.NET 4.0 with IIS?
- Does the application pool identity have read access to the source folder for your website?
- Have you stopped the Web Deployment Agent Service?

Configuring a Database Server for Web Deploy Publishing

This topic describes how to configure a SQL Server 2008 R2 database server to support web deployment and publishing.

The tasks described in this topic are common to every deployment scenario—it doesn't matter whether your web servers are configured to use the IIS Web Deployment Tool (Web Deploy) Remote Agent Service, the Web Deploy Handler, or offline deployment or your application is running on a single web server or a server farm. The way you deploy the database may change according to security requirements and other considerations. For example, you might deploy the database with or without sample data, and you might deploy user role mappings or configure them manually after deployment. However, the way you configure the database server remains the same.

Task Overview

You don't have to install any additional products or tools to configuring a database server to support web deployment. Assuming that your database server and your web server run on different machines, you simply need to:

- Permit SQL Server to communicate using TCP/IP.
- Allow SQL Server traffic through any firewalls.
- Give the web server machine account a SQL Server login.
- Map the machine account login to any required database roles.
- Give the account that will run the deployment a SQL Server login and database creator permissions.
- To support repeat deployments, map the deployment account login to the **db_owner** database role.

This topic will show you how to perform each of these procedures. The tasks and walkthroughs in this topic assume that you're starting with a default instance of SQL Server 2008 R2 running on Windows Server 2008 R2. Before you continue, ensure that:

- Windows Server 2008 R2 Service Pack 1 and all available updates are installed.
- The server is domain-joined.
- The server has a static IP address.
- SQL Server 2008 R2 Service Pack 1 and all available updates are installed.

The SQL Server instance only needs to include the **Database Engine Services** role, which is included automatically in any SQL Server installation. However, for ease of configuration and maintenance, we recommend that you include the **Management Tools – Basic** and **Management Tools – Complete** server roles.

Note: For more information on joining computers to a domain, see Joining Computers to the Domain and Logging On. For more information on configuring static IP addresses, see Configure a Static IP Address. For more information on installing SQL Server, see Installing SQL Server 2008 R2.

Enable Remote Access to SQL Server

SQL Server uses TCP/IP to communicate with remote computers. If your database server and your web server are on different machines, you need to:

- Configure SQL Server networking settings to allow communication over TCP/IP.
- Configure any hardware or software firewalls to allow TCP traffic (and in some cases User Datagram Protocol (UDP) traffic) on the ports that the SQL Server instance uses.

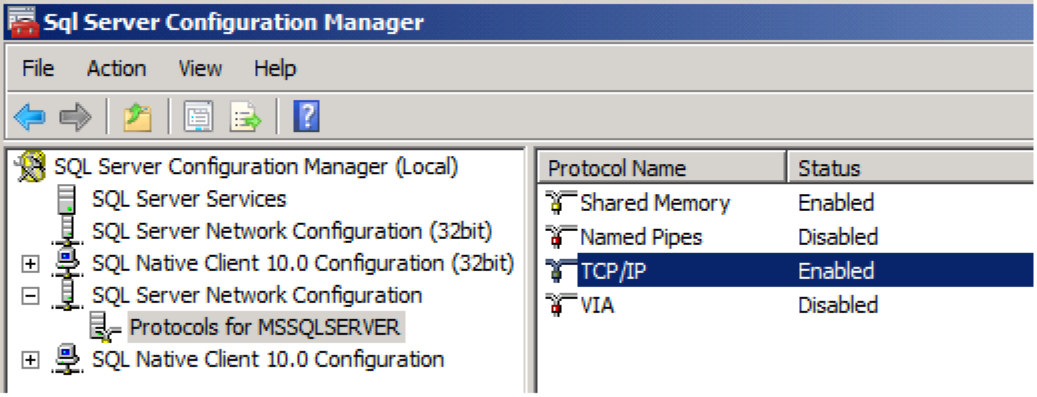
To enable SQL Server to communicate over TCP/IP, use SQL Server Configuration Manager to change the network configuration for your SQL Server instance.

To enable SQL Server to communicate using TCP/IP

1. On the **Start** menu, point to **All Programs**, click **Microsoft SQL Server 2008 R2**, click **Configuration Tools**, and then click **SQL Server Configuration Manager**.
2. In the tree view pane, expand **SQL Server Network Configuration**, and then click **Protocols for MSSQLSERVER**.

> **Note:** If you have installed multiple instances of SQL Server, you'll see a **Protocols for** *[instance name]* item for each instance. You need to configure network settings on an instance-by-instance basis.

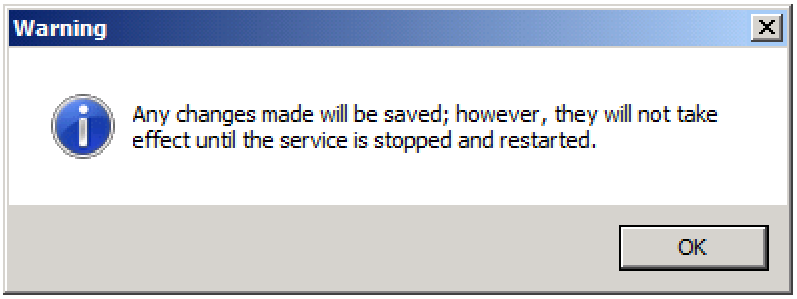
3. In the details pane, right-click the **TCP/IP** row, and then click **Enable**.



SQL Server Configuration Manager (Local)
SQL Server Services
SQL Server Network Configuration (32bit)
SQL Native Client 10.0 Configuration (32bit)
SQL Server Network Configuration
Protocols for MSSQLSERVER
SQL Native Client 10.0 Configuration

Protocol Name	Status
Shared Memory	Enabled
Named Pipes	Disabled
TCP/IP	Enabled
VIA	Disabled

4. In the **Warning** dialog box, click **OK**.

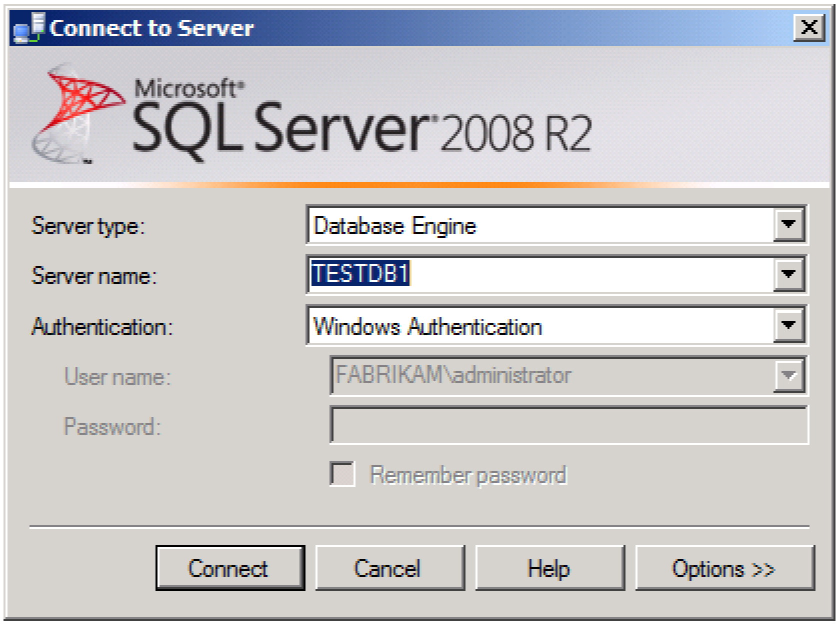


Any changes made will be saved; however, they will not take effect until the service is stopped and restarted.

You need to restart the MSSQLSERVER service before your new network configuration will take effect. You can do that at a command prompt, from the Services console, or from SQL Server Management Studio. In this procedure, you'll use SQL Server Management Studio.

5. Close SQL Server Configuration Manager.
6. On the **Start** menu, point to **All Programs**, click **Microsoft SQL Server 2008 R2**, and then click **SQL Server Management Studio**.

7. In the **Connect to Server** dialog box, in the **Server name** box, type the name of the database server, and then click **Connect**.



Server type: Database Engine

Server name: TESTDB1

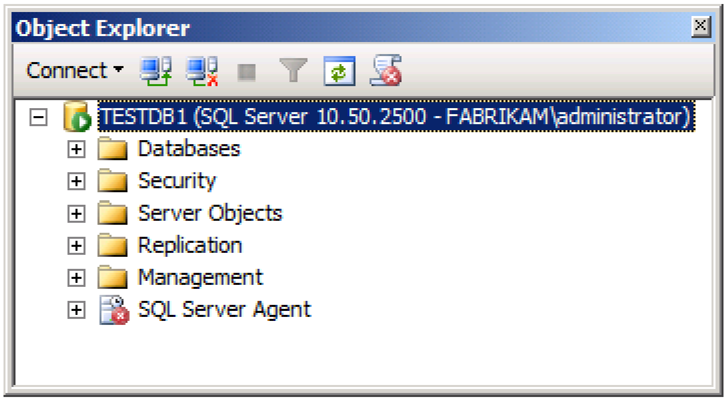
Authentication: Windows Authentication

User name: FABRIKAM\administrator

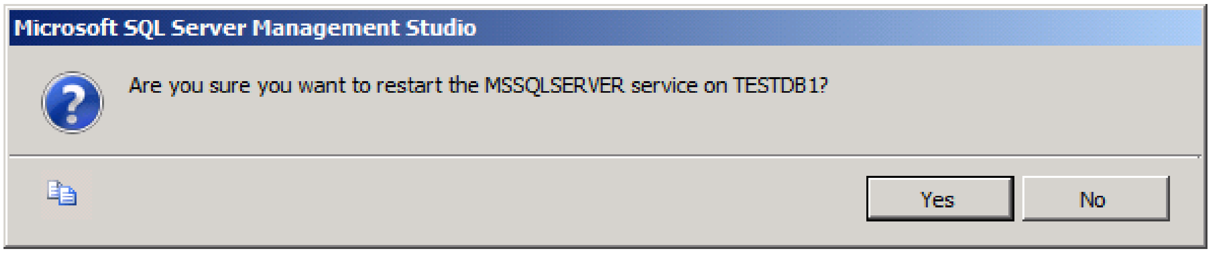
Password:

☐ Remember password

8. In the **Object Explorer** pane, right-click the parent server node (for example, **TESTDB1**), and then click **Restart**.



9. In the **Microsoft SQL Server Management Studio** dialog box, click **Yes**.



Are you sure you want to restart the MSSQLSERVER service on TESTDB1?

10. When the service has restarted, close SQL Server Management Studio.

To allow SQL Server traffic through a firewall, you first need to know which ports your SQL Server instance is using. This will depend on how the SQL Server instance was created and configured:

- A *default instance* of SQL Server listens for (and responds to) requests on TCP port 1433.
- A *named instance* of SQL Server listens for (and responds to) requests on a dynamically assigned TCP port.
- If the SQL Server Browser service is enabled, clients can query the service on UDP port 1434 to find out which TCP port to use for a particular SQL Server instance. However, this service is often disabled for security reasons.

Assuming that you're using a default instance of SQL Server, you need to configure your firewall to allow traffic.

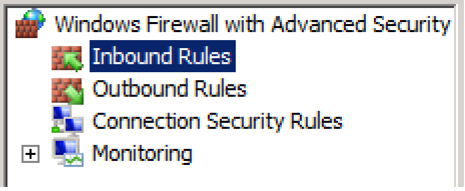
Direction	From Port	To Port	Port Type
Inbound	Any	1433	TCP
Outbound	1433	Any	TCP

Note: Technically, a client computer will use a randomly assigned TCP port between 1024 and 5000 to communicate with SQL Server, and you can restrict your firewall rules accordingly. For more information on SQL Server ports and firewalls, see TCP/IP port numbers required to communicate to SQL over a firewall and How to: Configure a Server to Listen on a Specific TCP Port (SQL Server Configuration Manager).

In most Windows Server environments, you'll likely have to configure Windows Firewall on the database server. By default, Windows Firewall allows all outbound traffic unless a rule specifically prohibits it. To enable your web server to reach your database, you need to configure an inbound rule that allows TCP traffic on the port number that the SQL Server instance uses. If you're using a default instance of SQL Server, you can use the next procedure to configure this rule.

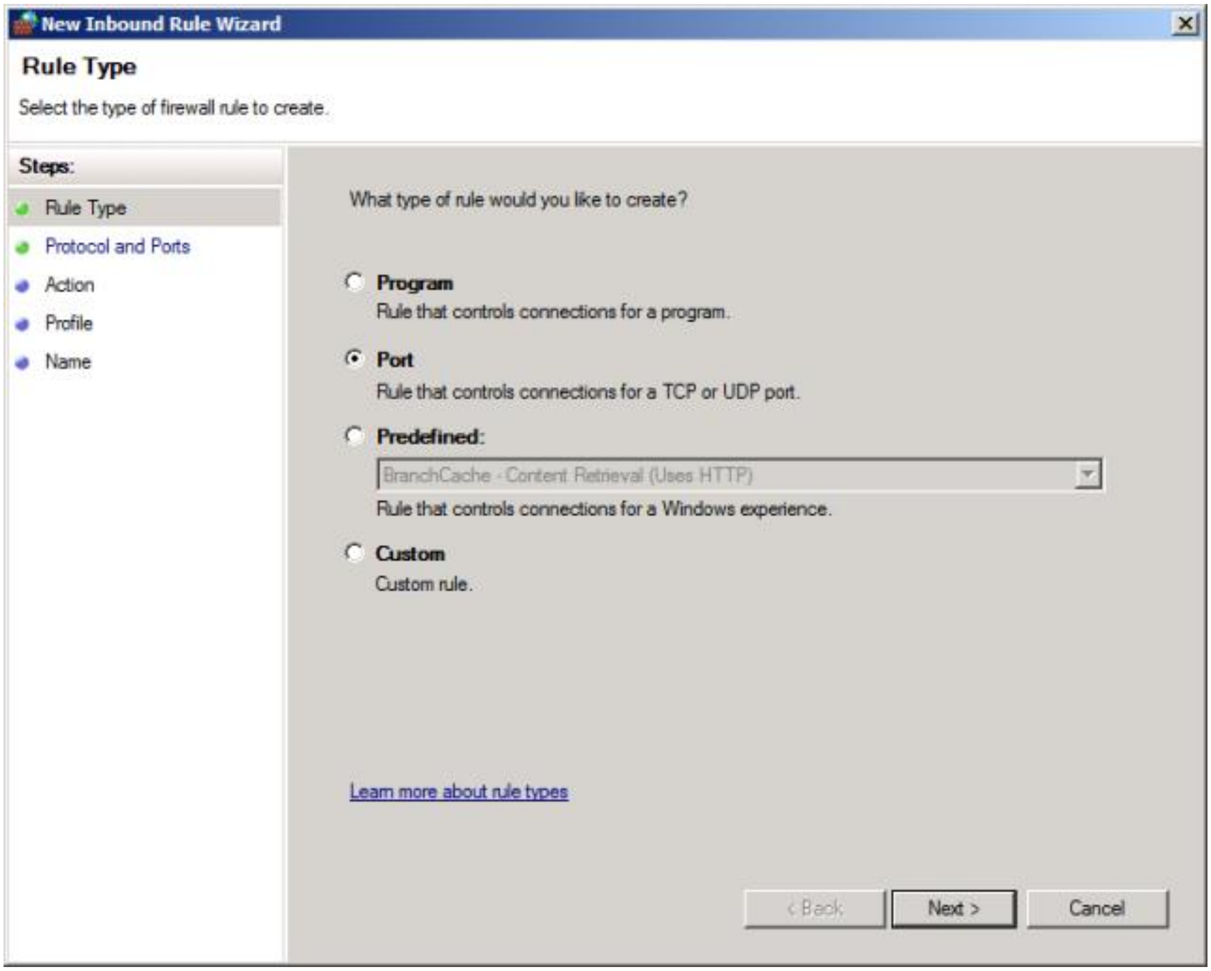
To configure Windows Firewall to allow communication with a default SQL Server instance

1. On the database server, on the **Start** menu, point to **Administrative Tools**, and then click **Windows Firewall with Advanced Security**.
2. In the tree view pane, click **Inbound Rules**.


Windows Firewall with Advanced Security
Inbound Rules
Outbound Rules
Connection Security Rules
Monitoring

3. In the **Actions** pane, under **Inbound Rules**, click **New Rule**.

4. In the New Inbound Rule Wizard, on the **Rule Type** page, select **Port**, and then click **Next**.



New Inbound Rule Wizard

Rule Type

Select the type of firewall rule to create.

Steps:

- Rule Type
- Protocol and Ports
- Action
- Profile
- Name

What type of rule would you like to create?

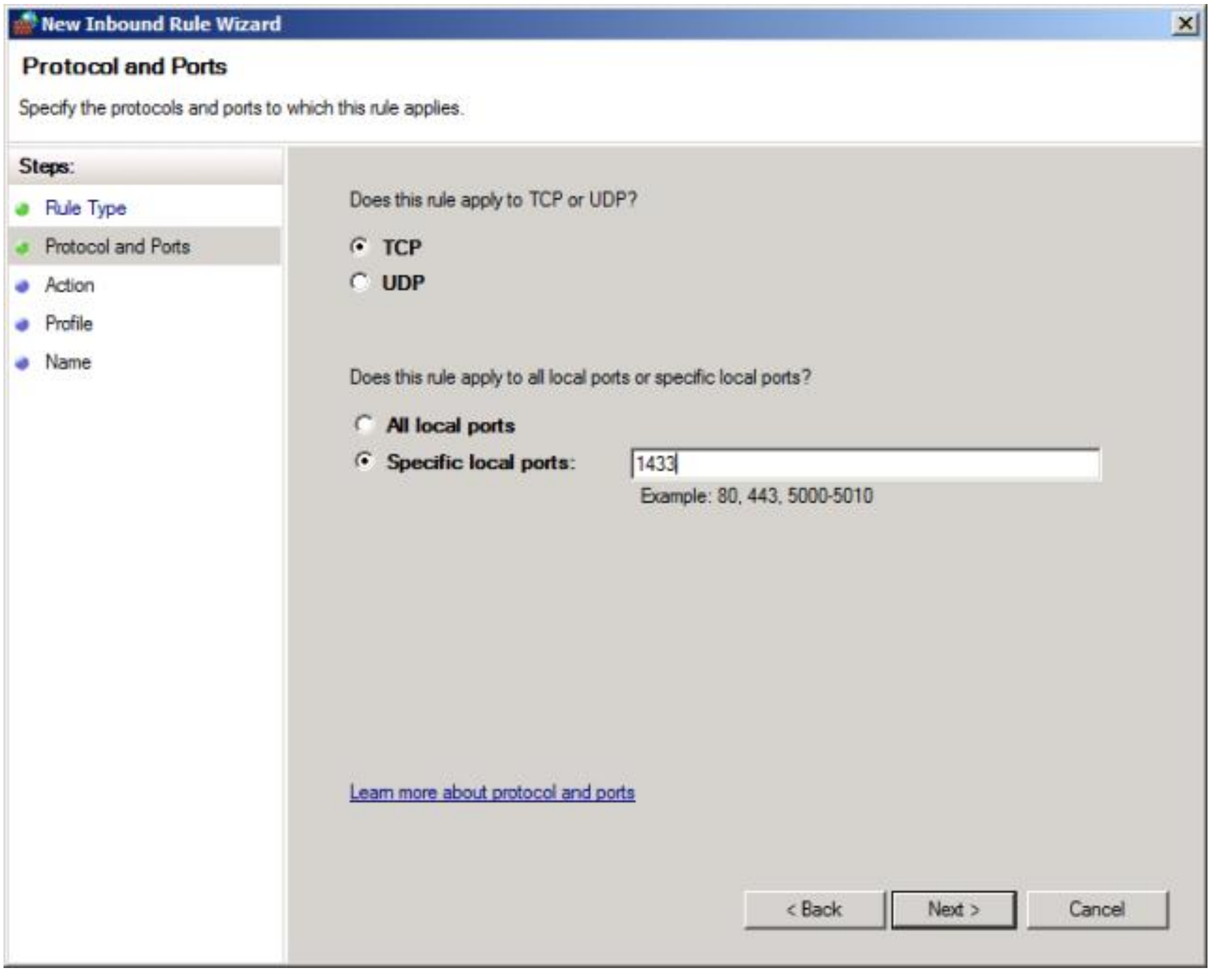
○ **Program**
Rule that controls connections for a program.

● **Port**
Rule that controls connections for a TCP or UDP port.

○ **Predefined:**
BranchCache - Content Retrieval (Uses HTTP)
Rule that controls connections for a Windows experience.

○ **Custom**
Custom rule.

5. On the **Protocol and Ports** page, ensure that **TCP** is selected, and in the **Specific local ports** box, type **1433**, and then click **Next**.


New Inbound Rule Wizard

Protocol and Ports

Specify the protocols and ports to which this rule applies.

Steps:

- Rule Type
- Protocol and Ports
- Action
- Profile
- Name

Does this rule apply to TCP or UDP?

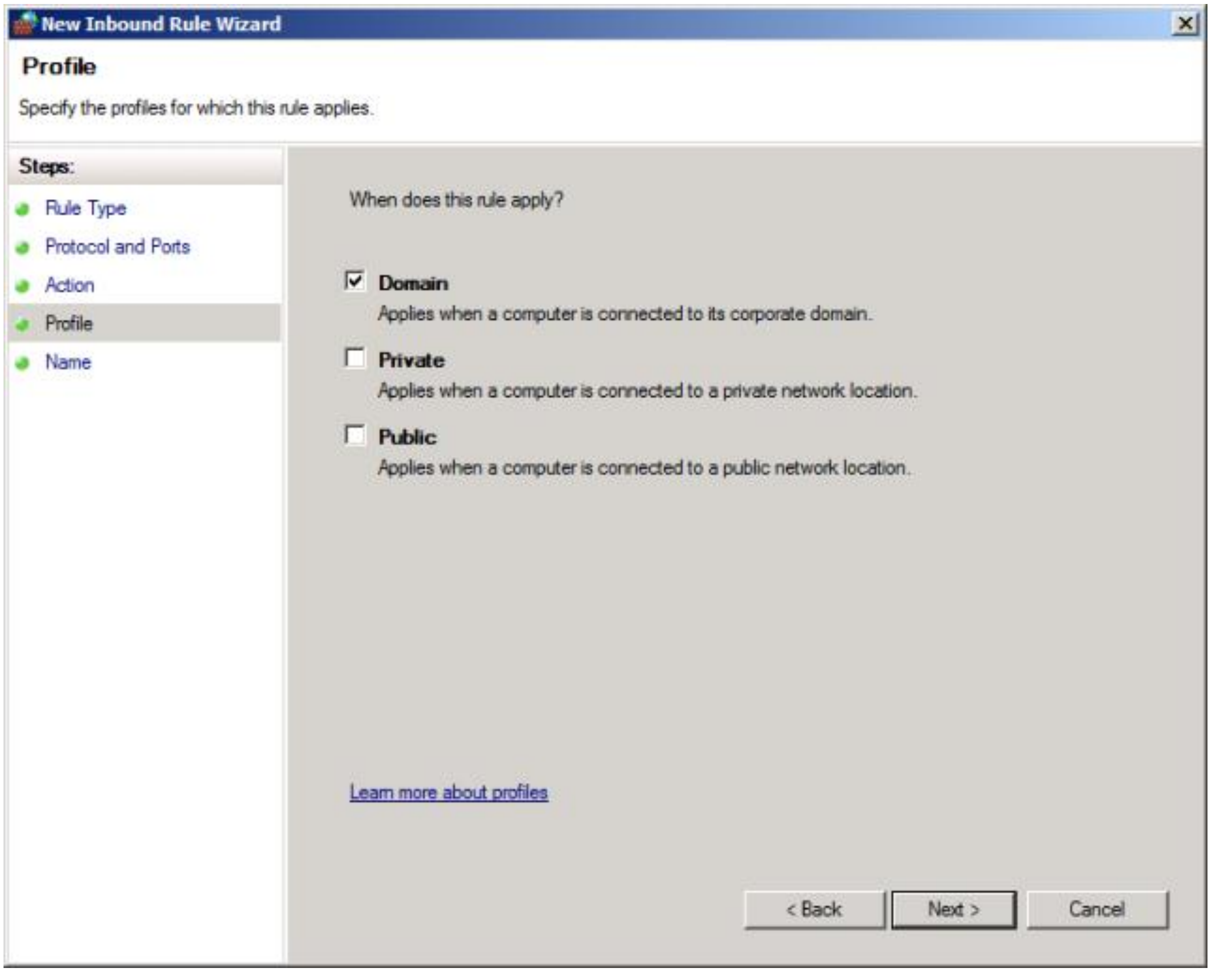
● TCP
○ UDP

Does this rule apply to all local ports or specific local ports?

○ All local ports
● Specific local ports: 1433

Example: 80, 443, 5000-5010

6. On the **Action** page, leave **Allow the connection** selected and click **Next**.
7. On the **Profile** page, leave **Domain** selected, clear the **Private** and **Public** check boxes, and then click **Next**.



Profile

Specify the profiles for which this rule applies.

When does this rule apply?

☑ **Domain**
Applies when a computer is connected to its corporate domain.

□ **Private**
Applies when a computer is connected to a private network location.

□ **Public**
Applies when a computer is connected to a public network location.

8. On the **Name** page, give the rule a suitably descriptive name (for example, **SQL Server default instance – network access**), and then click **Finish**.

For more information on configuring Windows Firewall for SQL Server, particularly if you need to communicate with SQL Server over non-standard or dynamic ports, see How to: Configure a Windows Firewall for Database Engine Access.

Configure Logins and Database Permissions

When you deploy a web application to Internet Information Services (IIS), the application runs using the identity of the application pool. In a domain environment, application pool identities use the machine account of the server on which they run to access network resources. Machine accounts take the form *[domain name]\[machine name]***$**—for example, **FABRIKAM\TESTWEB1$**. To allow your web application to access a database across the network, you need to:

- Add a login for the web server machine account to the SQL Server instance.
- Map the machine account login to any required database roles (typically **db_datareader** and **db_datawriter**).

If your web application is running on a server farm, rather than a single server, you'll need to repeat these procedures for every web server in the server farm.

Note: For more information on application pool identities and accessing network resources, see Application Pool Identities.

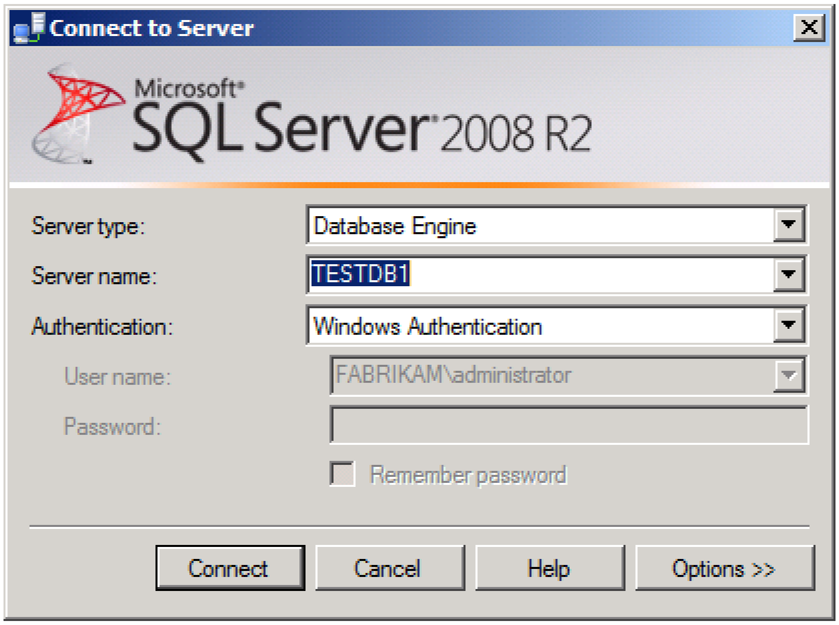
You can approach these tasks in various ways. To create the login, you can either:

- Create the login manually on the database server, using Transact-SQL or SQL Server Management Studio.
- Use a SQL Server 2008 Server Project in Visual Studio to create and deploy the login.

A SQL Server login is a server-level object, rather than a database-level object, so it's not dependent on the database you want to deploy. As such, you can create the login at any point, and the easiest approach is often to create the login manually on the database server before you start deploying databases. You can use the next procedure to create a login in SQL Server Management Studio.

To create a SQL Server login for the web server machine account

1. On the database server, on the **Start** menu, point to **All Programs**, click **Microsoft SQL Server 2008 R2**, and then click **SQL Server Management Studio**.
2. In the **Connect to Server** dialog box, in the **Server name** box, type the name of the database server, and then click **Connect**.



Server type: Database Engine

Server name: TESTDB1

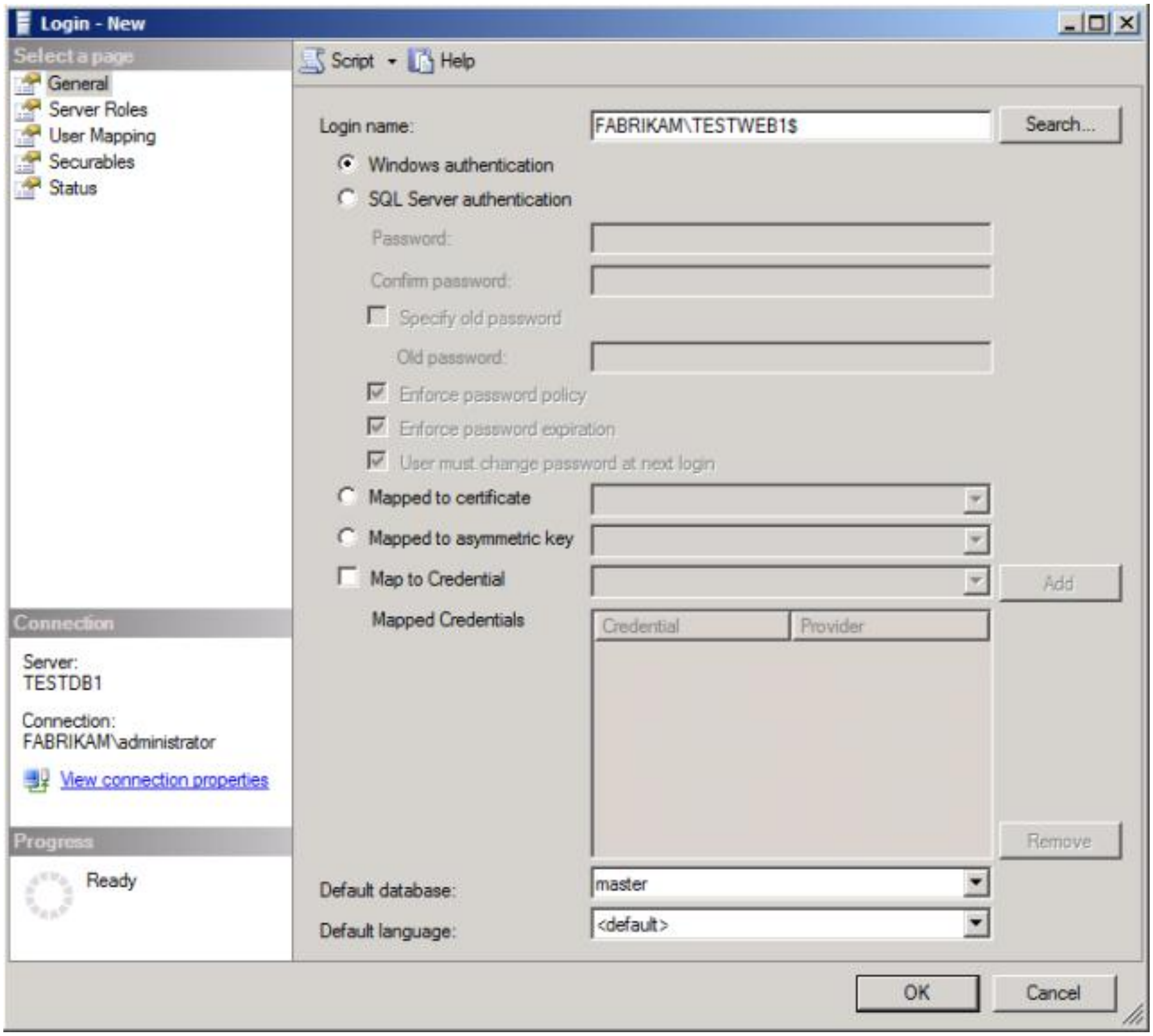
Authentication: Windows Authentication

User name: FABRIKAM\administrator

Password:

□ Remember password

3. In the **Object Explorer** pane, right-click **Security**, point to **New**, and then click **Login**.
4. In the **Login – New** dialog box, in the **Login name** box, type the name of your web server machine account (for example, **FABRIKAM\TESTWEB1$**).



Login name: FABRIKAM\TESTWEB1$

● Windows authentication

○ SQL Server authentication

Password:

Confirm password:

□ Specify old password

Old password:

☑ Enforce password policy

☑ Enforce password expiration

☑ User must change password at next login

○ Mapped to certificate

○ Mapped to asymmetric key

□ Map to Credential

Mapped Credentials

Credential	Provider

Default database: master

Default language: <default>

Server: TESTDB1

Connection: FABRIKAM\administrator

Ready

5. Click **OK**.

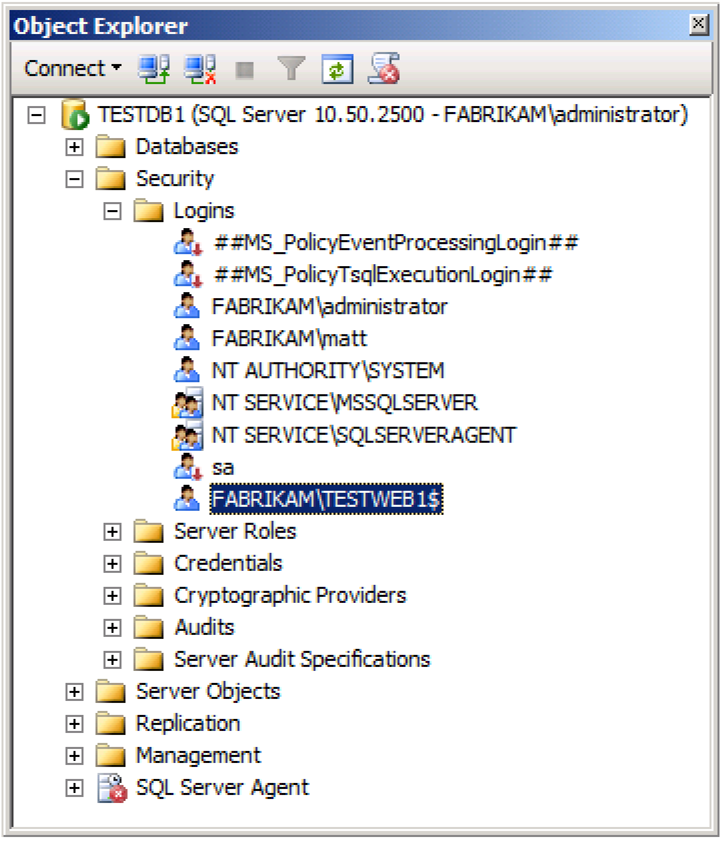
At this point, your database server is ready for Web Deploy publishing. However, any solutions you deploy won't work until you map the machine account login to the required database roles. Mapping the login to database roles requires a lot more thought, as you can't map roles until after you've deployed the database. To map the machine account login to the required database roles, you can either:

- Assign the database roles to the login manually, after you've deployed the database for the first time.
- Use a post-deployment script to assign the database roles to the login.

For more information on automating the creation of logins and database role mappings, see Deploying Database Role Memberships to Test Environments. Alternatively, you can use the next procedure to map the machine account login to the required database roles manually. Remember that you can't perform this procedure until *after* you've deployed the database.

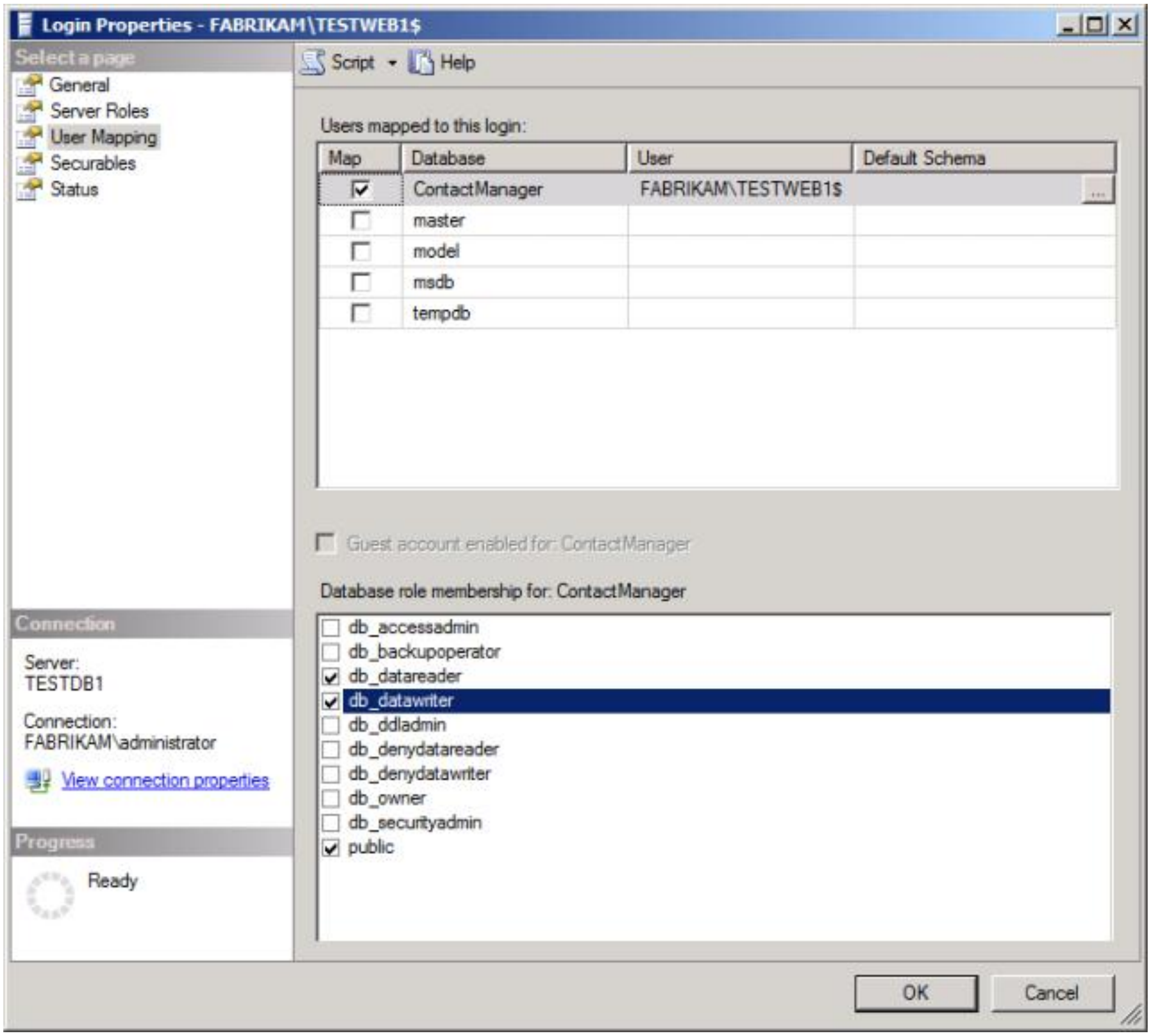
To map database roles to the web server machine account login

1. Open SQL Server Management Studio as before.
2. In the **Object Explorer** pane, expand the **Security** node, expand the **Logins** node, and then double-click the machine account login (for example, **FABRIKAM\TESTWEB1$**).



TESTDB1 (SQL Server 10.50.2500 - FABRIKAM\administrator)
- Databases
- Security
  - Logins
    - ##MS_PolicyEventProcessingLogin##
    - ##MS_PolicyTsqlExecutionLogin##
    - FABRIKAM\administrator
    - FABRIKAM\matt
    - NT AUTHORITY\SYSTEM
    - NT SERVICE\MSSQLSERVER
    - NT SERVICE\SQLSERVERAGENT
    - sa
    - FABRIKAM\TESTWEB1$
  - Server Roles
  - Credentials
  - Cryptographic Providers
  - Audits
  - Server Audit Specifications
- Server Objects
- Replication
- Management
- SQL Server Agent

3. In the **Login Properties** dialog box, click **User Mapping**.
4. In the **Users mapped to this login** table, select the name of your database (for example, **ContactManager**).
5. In the **Database role membership for:** *[database name]* list, select the permissions required. In the case of the Contact Manager sample solution, you must select the **db_datareader** and **db_datawriter** roles.



Users mapped to this login:

Map	Database	User	Default Schema
☑	ContactManager	FABRIKAM\TESTWEB1$	
□	master		
□	model		
□	msdb		
□	tempdb		

□ Guest account enabled for: ContactManager

Database role membership for: ContactManager

- □ db_accessadmin
- □ db_backupoperator
- ☑ db_datareader
- ☑ db_datawriter
- □ db_ddladmin
- □ db_denydatareader
- □ db_denydatawriter
- □ db_owner
- □ db_securityadmin
- ☑ public

Server:
TESTDB1

Connection:
FABRIKAM\administrator

View connection properties

Ready

6. Click **OK**.

While manually mapping database roles is often more than adequate for test environments, it's less desirable for automated or one-click deployments to staging or production environments. You can find more information on automating this kind of task using post-deployment scripts in Deploying Database Role Memberships to Test Environments.

> **Note:** For more information on server projects and database projects, see Visual Studio 2010 SQL Server Database Projects.

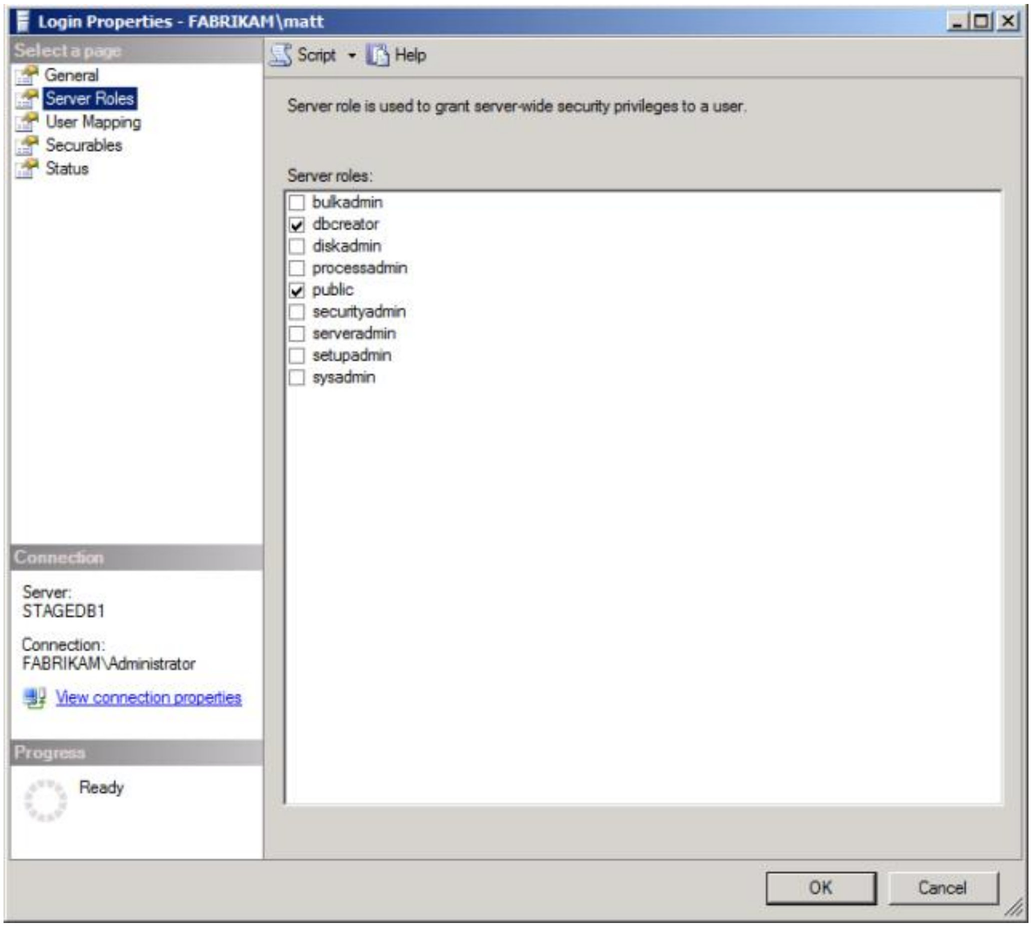
Configure Permissions for the Deployment Account

If the account that you'll use to run the deployment is not a SQL Server administrator, you'll also need to create a login for this account. In order to create the database, the account must be a member of the **dbcreator** server role or have equivalent permissions.

Note: When you use Web Deploy or VSDBCMD to deploy a database, you can use Windows credentials or SQL Server credentials (if your SQL Server instance is configured to support mixed mode authentication). The next procedure assumes that you want to use Windows credentials, but there's nothing stopping you from specifying a SQL Server user name and password in your connection string when you configure the deployment.

To set up permissions for the deployment account

1. Open SQL Server Management Studio as before.
2. In the **Object Explorer** pane, right-click **Security**, point to **New**, and then click **Login**.
3. In the **Login – New** dialog box, in the **Login name** box, type the name of your deployment account (for example, **FABRIKAM\matt**).
4. In the **Select a page** pane, click **Server Roles**.
5. Select **dbcreator**, and then click **OK**.



Server role is used to grant server-wide security privileges to a user.

Server roles:

☐ bulkadmin
☑ dbcreator
☐ diskadmin
☐ processadmin
☑ public
☐ securityadmin
☐ serveradmin
☐ setupadmin
☐ sysadmin

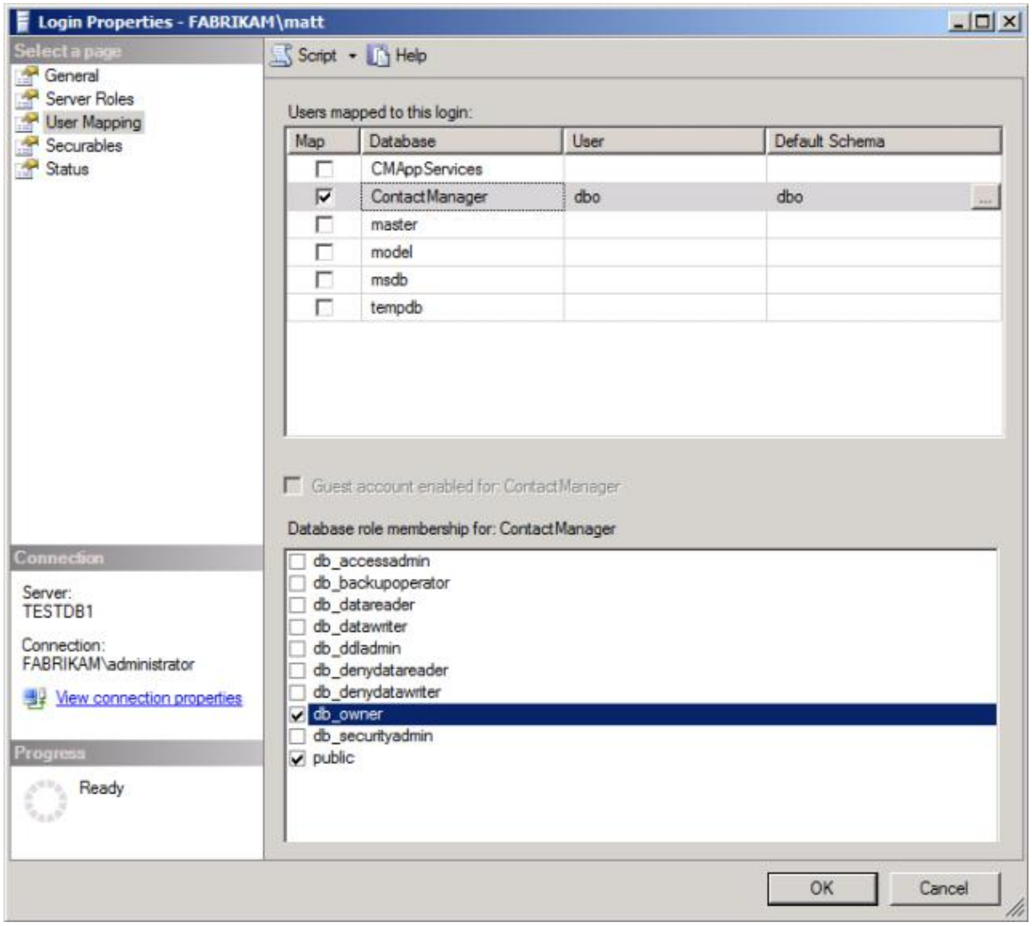
Server:
STAGEDB1

Connection:
FABRIKAM\Administrator

To support subsequent deployments, you'll also need to add the deploying account to the **db_owner** role on the database after the first deployment. This is because on subsequent deployments you're modifying the schema of an existing database, rather than creating a new database. As described in the previous section, you can't add a user to a database role until you've created the database, for obvious reasons.

To map the deployment account login to the db_owner database role

1. Open SQL Server Management Studio as before.
2. In the **Object Explorer** window, expand the **Security** node, expand the **Logins** node, and then double-click the machine account login (for example, **FABRIKAM\matt**).
3. In the **Login Properties** dialog box, click **User Mapping**.
4. In the **Users mapped to this login** table, select the name of your database (for example, **ContactManager**).
5. In the **Database role membership for:** *[database name]* list, select the **db_owner** role.



Users mapped to this login:

Map	Database	User	Default Schema
□	CMAppServices		
☑	ContactManager	dbo	dbo
□	master		
□	model		
□	msdb		
□	tempdb		

□ Guest account enabled for: ContactManager

Database role membership for: ContactManager

- □ db_accessadmin
- □ db_backupoperator
- □ db_datareader
- □ db_datawriter
- □ db_ddladmin
- □ db_denydatareader
- □ db_denydatawriter
- ☑ db_owner
- □ db_securityadmin
- ☑ public

Server:
TESTDB1

Connection:
FABRIKAM\administrator

Ready

6. Click **OK**.

Conclusion

Your database server should now be ready to accept remote database deployments and to allow remote IIS web servers to access your databases. Before you attempt to deploy and use databases, you may want to check these key points:

- Have you configured SQL Server to accept remote TCP/IP connections?
- Have you configured any firewalls to permit SQL Server traffic?
- Have you created a machine account login for every web server that will access SQL Server?
- Does your database deployment include a script to create user role mappings, or do you need to create these manually after you deploy the database for the first time?
- Have you created a login for the deployment account and added it to the **dbcreator** server role?

Further Reading

For guidance on deploying database projects, see Deploying Database Projects. For guidance on creating database role memberships by running a post-deployment script, see Deploying Database Role Memberships to Test Environments. For guidance on how to meet the unique deployment challenges that membership databases pose, see Deploying Membership Databases to Enterprise Environments.

Creating a Server Farm with the Web Farm Framework

This topic describes how to use the Web Farm Framework (WFF) 2.0 to create and configure a web server farm from a collection of servers.

Why Should You Create a Server Farm?

WFF lets you synchronize web platform products and components, web applications, websites, and configuration settings across multiple load-balanced web servers. In scenarios where you need more than one web server, like staging and production environments, this can vastly simplify your deployment and configuration process. You can deploy a web application to a single server—the *primary server*—and WFF will automatically replicate that web application on all the other web servers in the server farm.

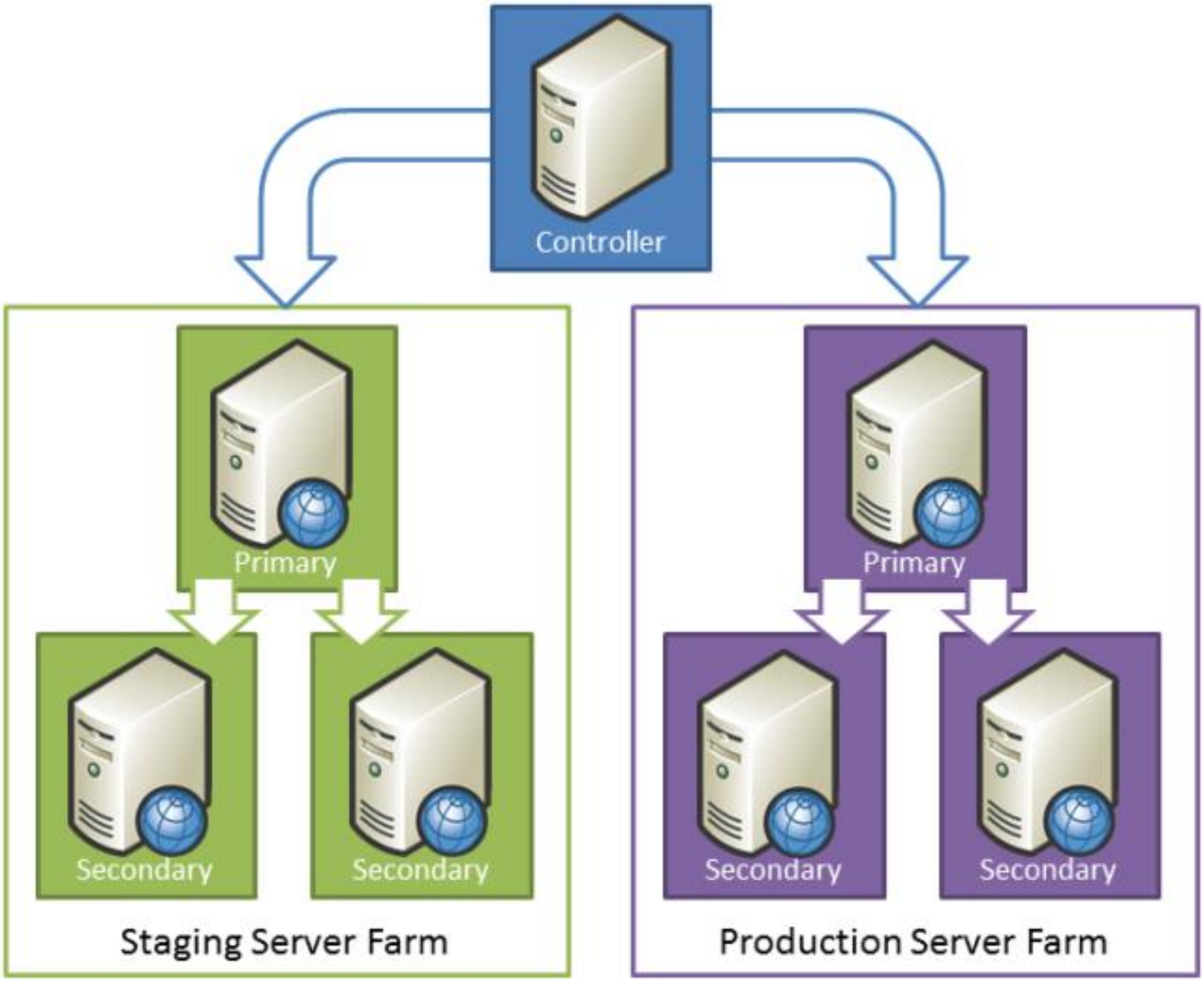
Understanding the Web Farm Framework

You can use WFF 2.0 to provision, manage, and deploy content to a group of web servers. A WFF deployment consists of three key server roles:

- The *controller server*. You use this server to create and configure WFF server farms. The controller server manages the synchronization of web platform components, configuration settings, and applications between the web servers in a server farm. You install WFF 2.0 on the controller server, and the controller server will in turn install the WFF agent on each of the servers in a server farm. The controller server does not conceptually belong to any WFF server farm, and a single controller server can manage multiple server farms. In this scenario, you use a single WFF controller server to create and manage the staging server farm and the production server farm.

- The *primary server*. Each WFF server farm includes a single primary server. When you install web platform components or deploy applications to the primary server, the WFF synchronizes your changes to all the other servers in the server farm.
- The *secondary server*. Each WFF server farm includes one or more secondary servers. Any changes you make to the primary server are replicated to every secondary server within the server farm.

This shows how these server roles relate to the Fabrikam, Inc. staging and production environments:



In this scenario, the staging environment and the production environment are both configured as WFF server farms. A single WFF controller server manages both farms. Within each server farm, any changes to the primary server are replicated to every secondary server.

Before you start to configure your staging and production environments, we recommend that you read these articles to familiarize yourself with the key concepts of WFF 2.0:

- Overview of the Web Farm Framework 2.0 for IIS 7
- Setting up a Server Farm with the Web Farm Framework 2.0 for IIS 7
- System and Platform Requirements for the Web Farm Framework 2.0 for IIS 7

Task Overview

To complete the tasks and walkthroughs in this topic, you'll need at least three servers—one WFF controller, one primary web server for the server farm, and one or more secondary web servers for the server farm. You can add more secondary servers to a WFF server farm at any time. At a high level, to create and configure a WFF server farm for your staging or production environment you'll need to:

- Create a controller server by installing Internet Information Services (IIS) 7.5 and WFF 2.0.
- Prepare primary and secondary servers by creating a common administrator account and configuring firewall exceptions.
- Configure the server farm by using IIS Manager on the controller server.
- Configure load balancing using IIS Application Request Routing (ARR) or an alternative load-balancing technology.

The tasks and walkthroughs in this topic assume that you're starting with clean server builds running Windows Server 2008 R2. Before you begin, for each server, ensure that:

- Windows Server 2008 R2 Service Pack 1 and all available updates are installed.
- The server is domain-joined.
- The server has a static IP address.

Note: For more information on joining computers to a domain, see Joining Computers to the Domain and Logging On. For more information on configuring static IP addresses, see Configure a Static IP Address.

Create the WFF Controller Server

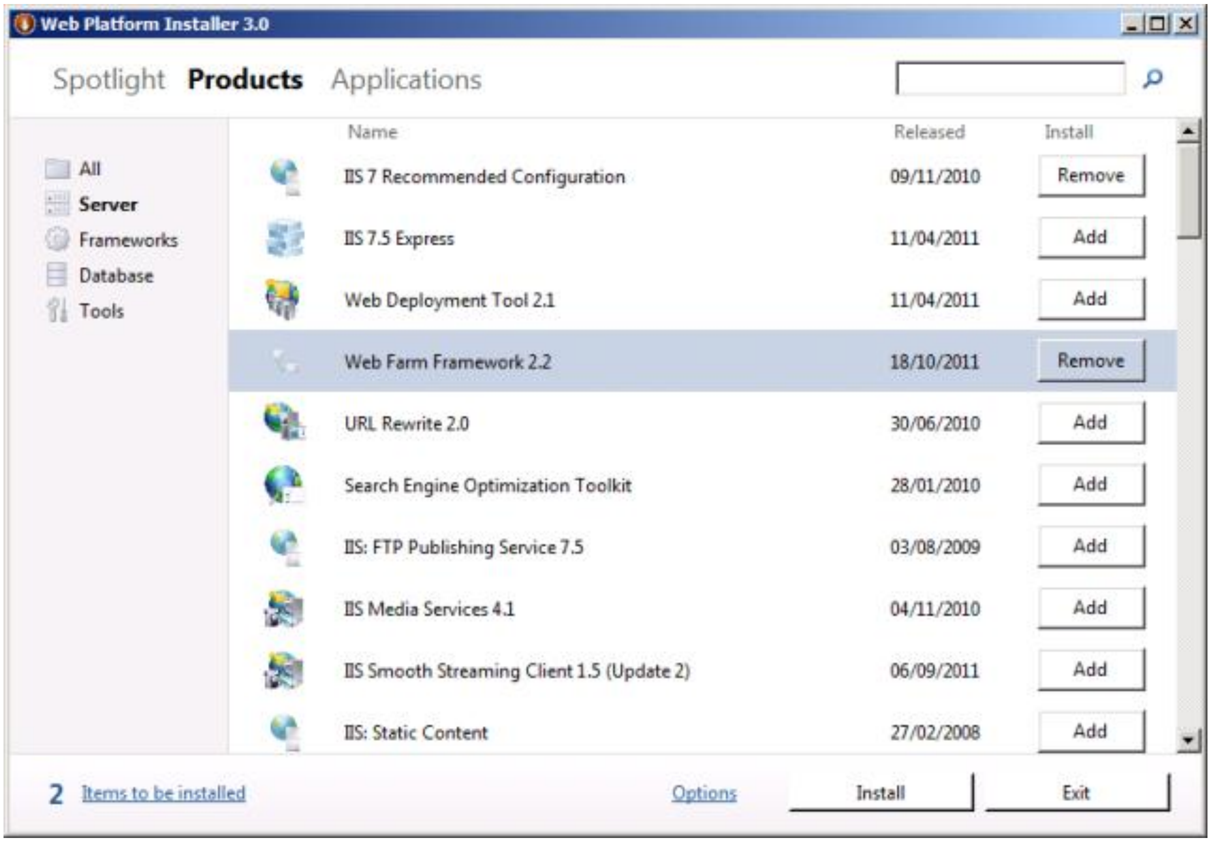
To create a WFF controller server, you'll need to install both IIS 7 or later and WFF 2.0 or later. Under the covers, WFF uses the IIS Web Deployment Tool (Web Deploy) 2.x to synchronize the servers in your farm. If you use the Web Platform Installer to install WFF, the installer will automatically download and install Web Deploy for you.

To create the WFF controller server

1. Download and install the Web Platform Installer.
2. At the top of the **Web Platform Installer 3.0** window, click **Products**.
3. On the left side of the window, in the navigation pane, click **Server**.

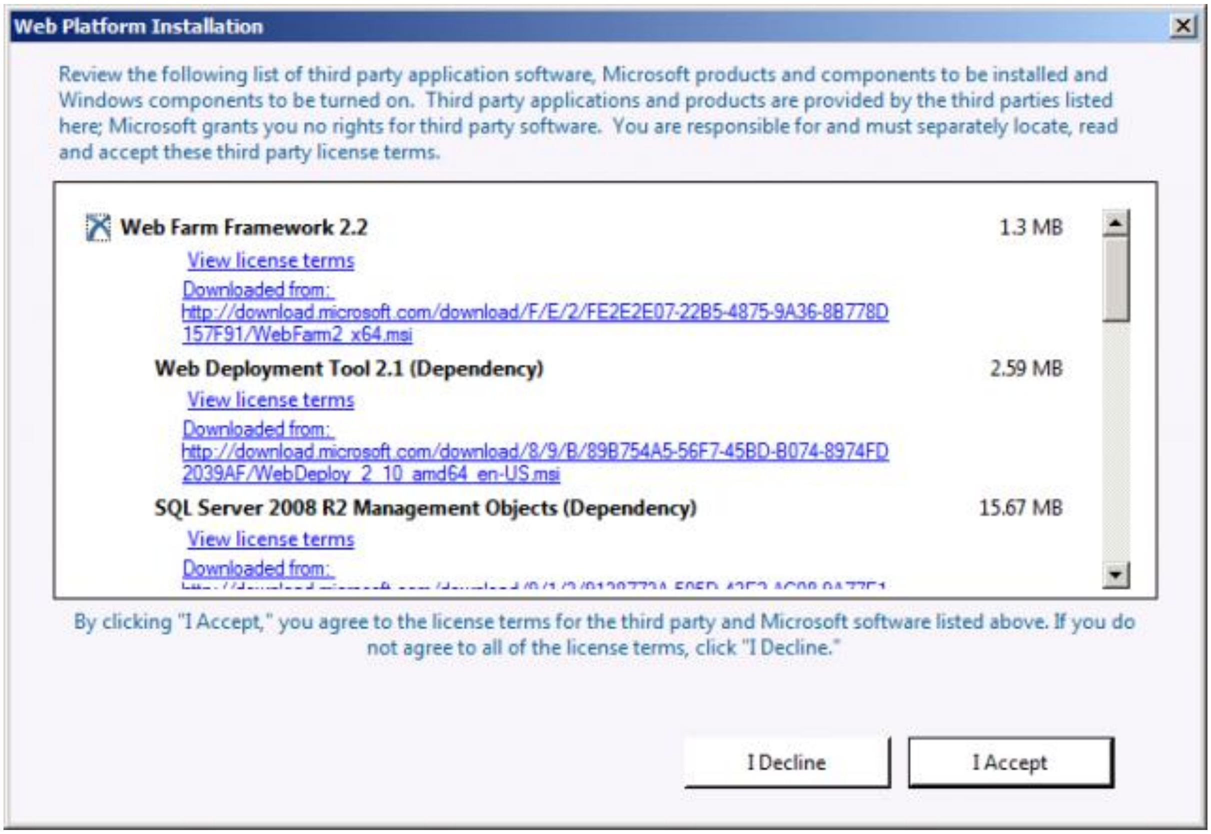
4. In the **IIS 7 Recommended Configuration** row, click **Add**.

5. In the **Web Farm Framework 2.***x* row, click **Add**.



Name	Released	Install
IIS 7 Recommended Configuration	09/11/2010	Remove
IIS 7.5 Express	11/04/2011	Add
Web Deployment Tool 2.1	11/04/2011	Add
Web Farm Framework 2.2	18/10/2011	Remove
URL Rewrite 2.0	30/06/2010	Add
Search Engine Optimization Toolkit	28/01/2010	Add
IIS: FTP Publishing Service 7.5	03/08/2009	Add
IIS Media Services 4.1	04/11/2010	Add
IIS Smooth Streaming Client 1.5 (Update 2)	06/09/2011	Add
IIS: Static Content	27/02/2008	Add

6. Click **Install**. Notice that the Web Platform Installer has added the Web Deployment Tool, along with various other dependencies, to the installation list.



Review the following list of third party application software, Microsoft products and components to be installed and Windows components to be turned on. Third party applications and products are provided by the third parties listed here; Microsoft grants you no rights for third party software. You are responsible for and must separately locate, read and accept these third party license terms.

Web Farm Framework 2.2 — 1.3 MB
View license terms
Downloaded from:
http://download.microsoft.com/download/F/E/2/FE2E2E07-22B5-4875-9A36-8B778D157F91/WebFarm2_x64.msi

Web Deployment Tool 2.1 (Dependency) — 2.59 MB
View license terms
Downloaded from:
http://download.microsoft.com/download/8/9/B/89B754A5-56F7-45BD-B074-8974FD2039AF/WebDeploy_2_10_amd64_en-US.msi

SQL Server 2008 R2 Management Objects (Dependency) — 15.67 MB
View license terms
Downloaded from:

By clicking "I Accept," you agree to the license terms for the third party and Microsoft software listed above. If you do not agree to all of the license terms, click "I Decline."

7. Review the license terms, and if you consent to the terms, click **I Accept**.
8. When the installation is complete, click **Finish**, and then close the **Web Platform Installer 3.0** window.

Configure the Primary and Secondary Servers

Before you create a WFF server farm, you should complete some preparation tasks on the web servers that will make up the farm:

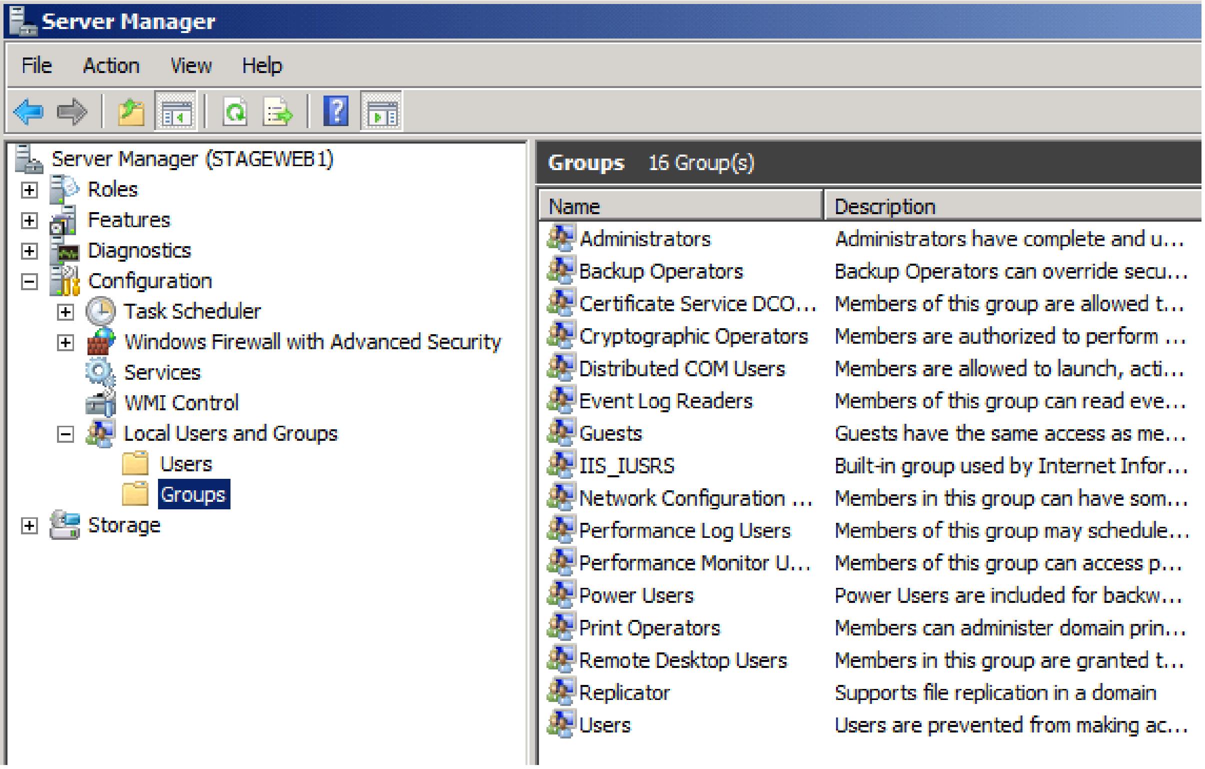
- Add firewall exceptions to allow the **Core Networking**, **Remote Administration**, and **File and Printer Sharing** features to communicate with the WFF controller server.
- Create a domain account (for example, **FABRIKAM\stagingfarm**) in Active Directory and add it to the local administrators group on each server. You'll use this account as the server farm administrator account when you create the server farm.

For more information on how to configure these firewall exceptions in Windows Firewall, see System and Platform Requirements for the Web Farm Framework 2.0 for IIS 7. For other firewall systems, consult your product documentation.

You can use the next procedure to add a domain account to the local administrators group in Windows Server 2008 R2. You should perform this procedure on every server that you want to add to the server farm—in other words, add the same domain account to the local administrators group on the primary server and on each secondary server.

To add a domain account to the local administrators group

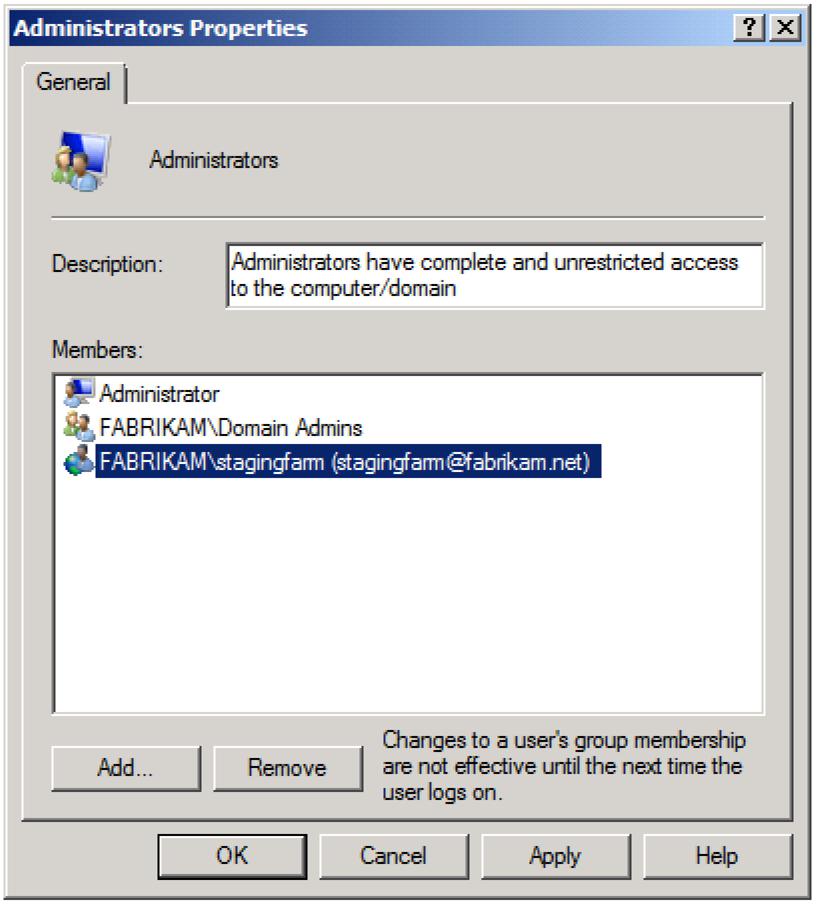
1. On the **Start** menu, point to **Administrative Tools**, and then click **Server Manager**.
2. In the **Server Manager** window, in the tree view pane, expand **Configuration**, expand **Local Users and Groups**, and then click **Groups**.



Groups 16 Group(s)

Name	Description
Administrators	Administrators have complete and u…
Backup Operators	Backup Operators can override secu…
Certificate Service DCO…	Members of this group are allowed t…
Cryptographic Operators	Members are authorized to perform …
Distributed COM Users	Members are allowed to launch, acti…
Event Log Readers	Members of this group can read eve…
Guests	Guests have the same access as me…
IIS_IUSRS	Built-in group used by Internet Infor…
Network Configuration …	Members in this group can have som…
Performance Log Users	Members of this group may schedule…
Performance Monitor U…	Members of this group can access p…
Power Users	Power Users are included for backw…
Print Operators	Members can administer domain prin…
Remote Desktop Users	Members in this group are granted t…
Replicator	Supports file replication in a domain
Users	Users are prevented from making ac…

3. In the **Groups** pane, double-click **Administrators**.
4. In the **Administrators Properties** dialog box, click **Add**.
5. In the **Select Users, Computers, Service Accounts, or Groups** dialog box, type (or browse) to your domain account (for example, **FABRIKAM\stagingfarm**), and then click **OK**.



Administrators Properties

Administrators

Description: Administrators have complete and unrestricted access to the computer/domain

Members:

- Administrator
- FABRIKAM\Domain Admins
- FABRIKAM\stagingfarm (stagingfarm@fabrikam.net)

Changes to a user's group membership are not effective until the next time the user logs on.

6. In the **Administrators Properties** dialog box, click **OK**.

Your servers are now ready to be added to a server farm. In the case of the primary server, you can configure the server to meet your application requirements before or after you create the server farm—in both cases, the WFF will synchronize the servers by deploying the same products, components, or configuration to your secondary servers. For the sake of simplicity, this tutorial assumes that you'll configure the primary server when you've finished creating the server farm.

Create the WFF Server Farm

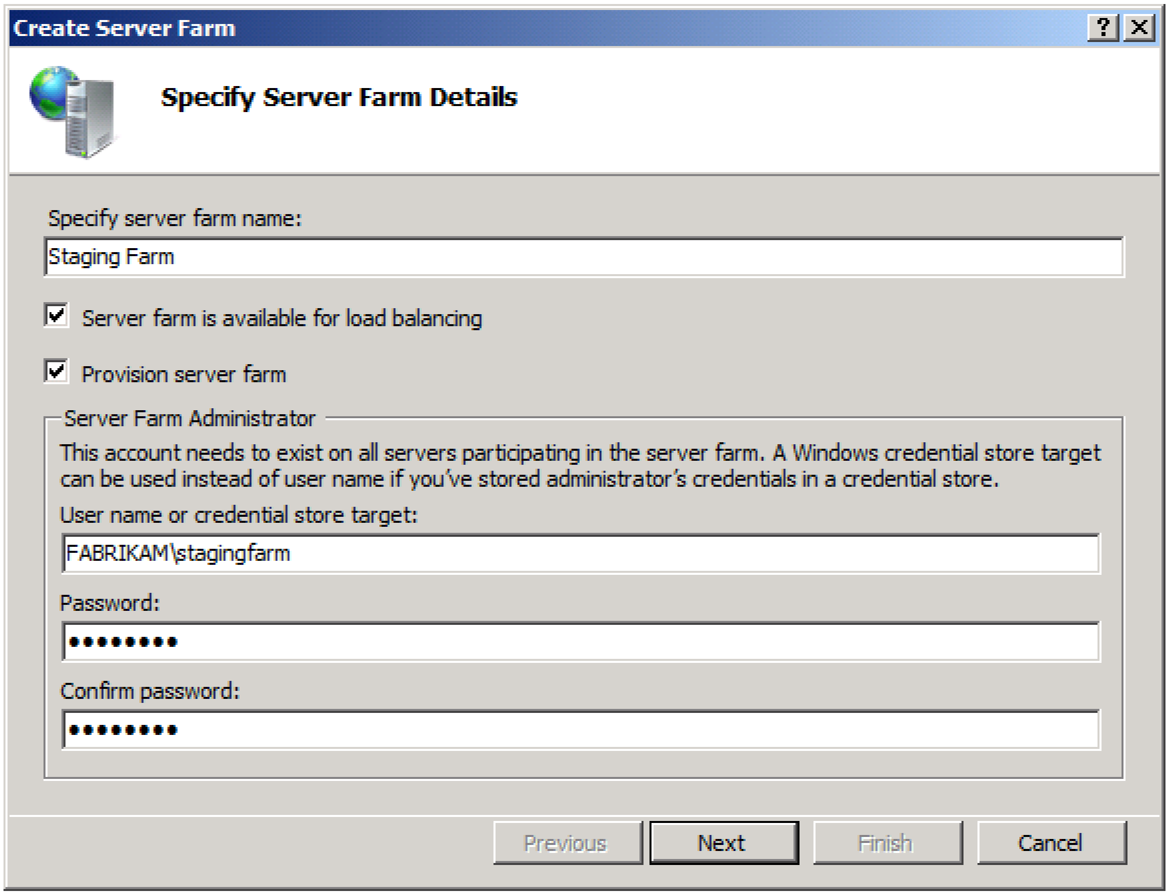
At this point, all your servers are ready to be added to a WFF server farm:

- You've installed WFF on the controller server.
- You've configured firewall exceptions on your primary and secondary web servers.
- You've added a domain account to the local administrators group on your primary and secondary web servers.

The next step is to create the server farm in WFF. You can do this from IIS Manager on the WFF controller server.

To create a WFF server farm

1. On the WFF controller server, on the **Start** menu, point to **Administrative Tools**, and then click **Internet Information Services (IIS) Manager**.
2. In the **Connections** pane, expand the local server node, right-click **Server Farms**, and then click **Create Server Farm**.
3. In the **Create Server Farm** dialog box, type a meaningful name for the server farm (for example, **Staging Farm**), and then select **Provision server farm**.
4. Type the user name and password of the domain account that you added to the local administrators group on each server.



Specify server farm name:

Staging Farm

☑ Server farm is available for load balancing

☑ Provision server farm

Server Farm Administrator

This account needs to exist on all servers participating in the server farm. A Windows credential store target can be used instead of user name if you've stored administrator's credentials in a credential store.

User name or credential store target:

FABRIKAM\stagingfarm

Password:

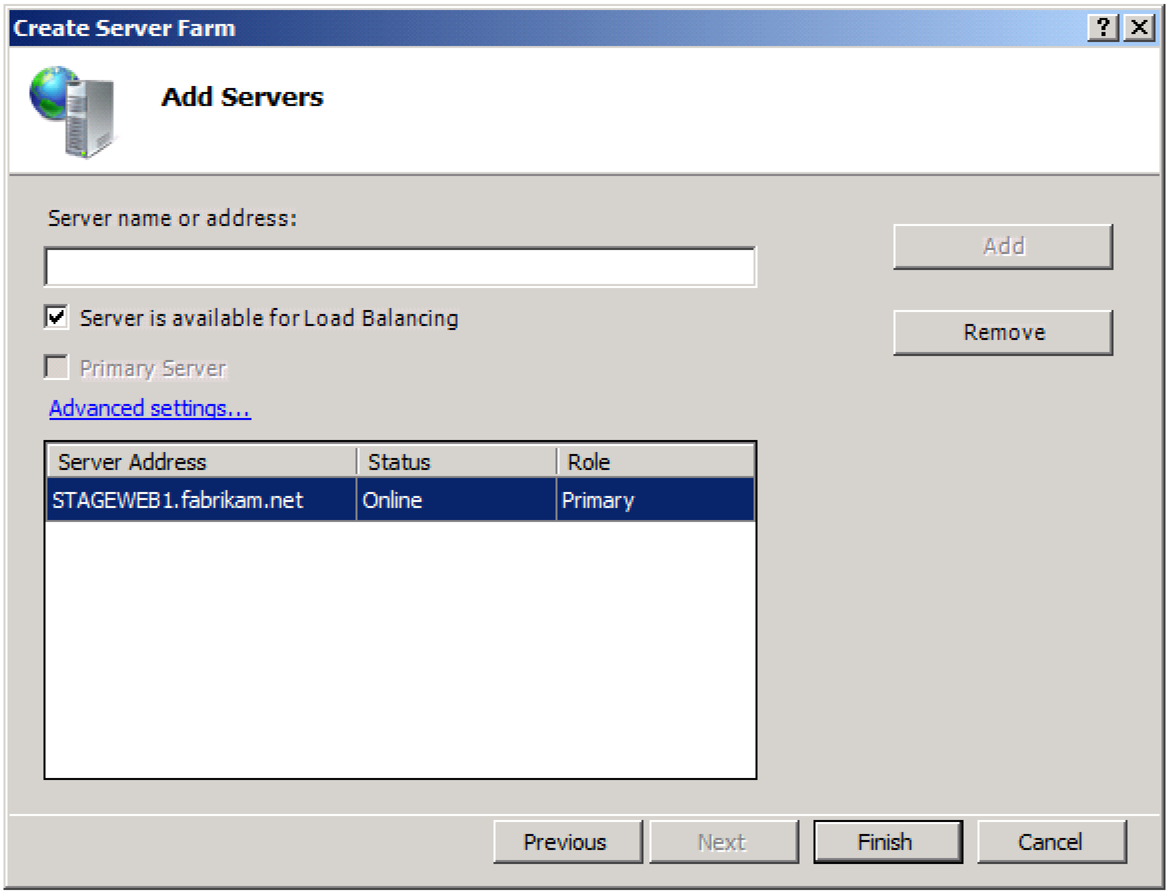
••••••••

Confirm password:

••••••••

5. Click **Next**.
6. On the **Add Servers** page, type the fully qualified domain name (FQDN) of the primary server, select **Primary Server**, and then click **Add**.

   At this point, WFF will attempt to contact the primary server using the credentials you provided. If the connection succeeds, the primary server will be added to the table on the **Add Servers** page.



Server name or address:

☑ Server is available for Load Balancing

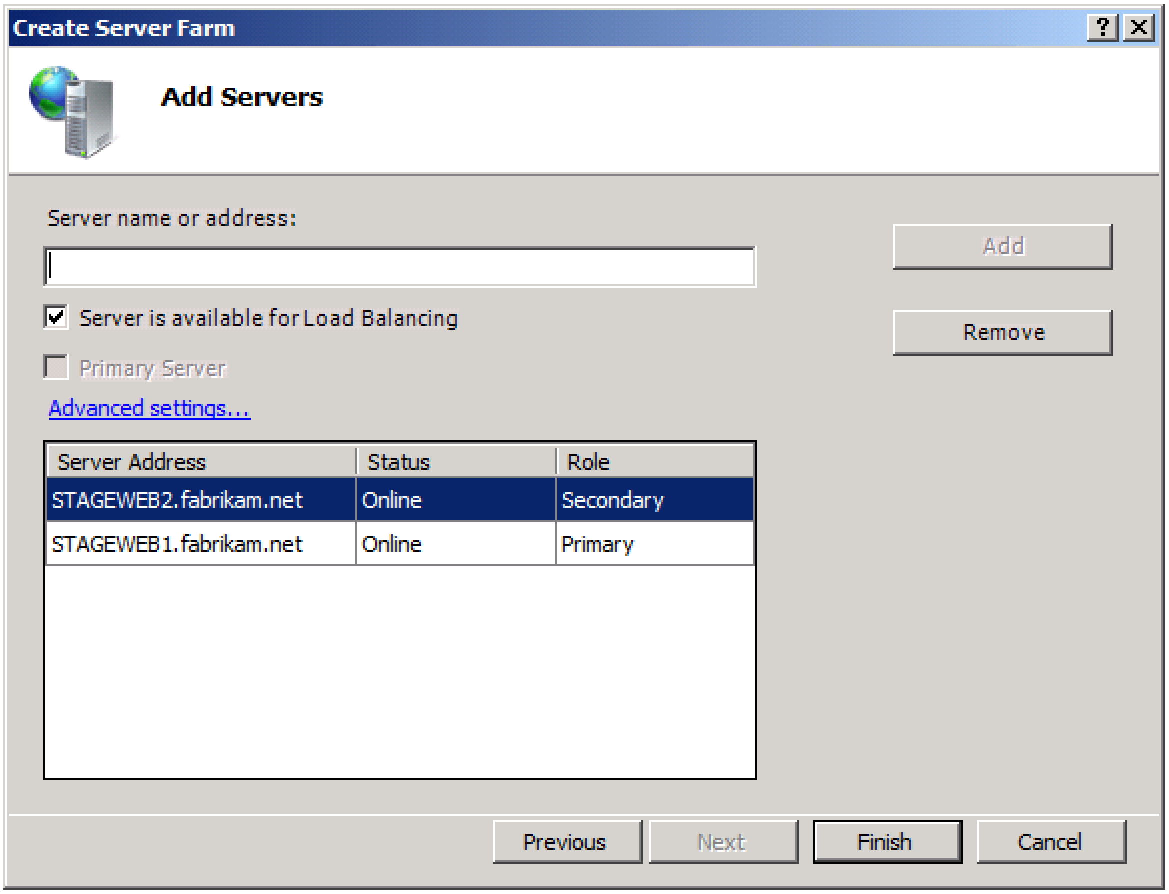
□ Primary Server

Advanced settings...

Server Address	Status	Role
STAGEWEB1.fabrikam.net	Online	Primary

Note: You might have noticed that **Server is available for Load Balancing** is selected by default. WFF uses the IIS ARR module to implement load balancing and thereby distribute requests across the web servers in your server farm. In most scenarios, you'd only clear the **Server is available for Load Balancing** option if you wanted to use a third-party load balancing solution instead.

7. On the **Add Servers** page, type the FQDN of your first secondary server, and then click **Add**.



Server name or address:

☑ Server is available for Load Balancing

□ Primary Server

Server Address	Status	Role
STAGEWEB2.fabrikam.net	Online	Secondary
STAGEWEB1.fabrikam.net	Online	Primary

8. Repeat step 7 for any additional secondary servers in your farm, and then click **Finish**.

Your WFF server farm is now up and running. Any web platform products or components that you install on the primary server, and any web applications or content that you deploy to the primary server, will be automatically provisioned on all your secondary servers.

WFF is a broad and complex topic, and you can learn more about it on the Microsoft Web Farm Framework 2.0 for IIS 7 website. For the time being, however, there are two features areas that you need to be aware of:

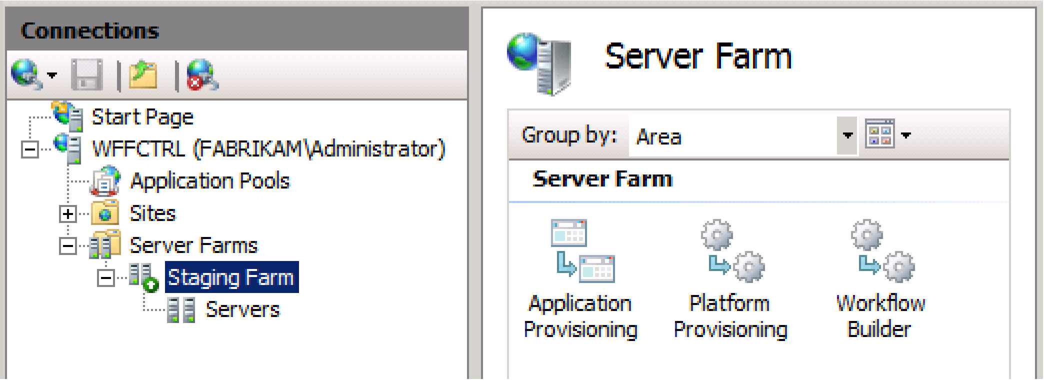
- *Application provisioning* is the process that replicates content from the primary server, like web applications and configuration settings, across all the secondary servers in the server farm. For example, if you deploy the Contact Manager sample solution to your primary staging server, the WFF application provisioning process will deploy this solution to all your secondary staging servers. By default, the application provisioning process runs every 30 seconds.
- *Platform provisioning* is the process that synchronizes web platform products and components from the primary server to all the secondary servers in the server farm. For example, if you install ASP.NET MVC 3 on your primary staging server, the platform provisioning process will use

the Web Platform Installer to install ASP.NET MVC 3 on all your secondary staging servers. By default, the platform provisioning process runs every five minutes.

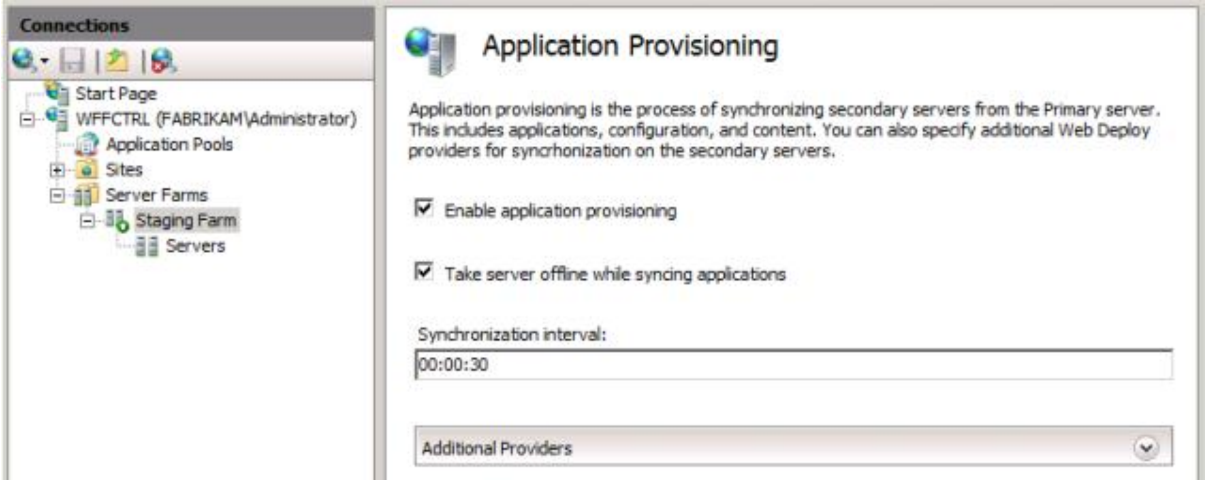
You can manage basic application and platform provisioning settings from IIS Manager on your WFF controller server.

Explore application and platform provisioning settings

1. In IIS Manager, in the **Connections** pane, select your server farm.



2. In the **Server Farm** pane, double-click **Application Provisioning**.



Application provisioning is the process of synchronizing secondary servers from the Primary server. This includes applications, configuration, and content. You can also specify additional Web Deploy providers for synchronization on the secondary servers.

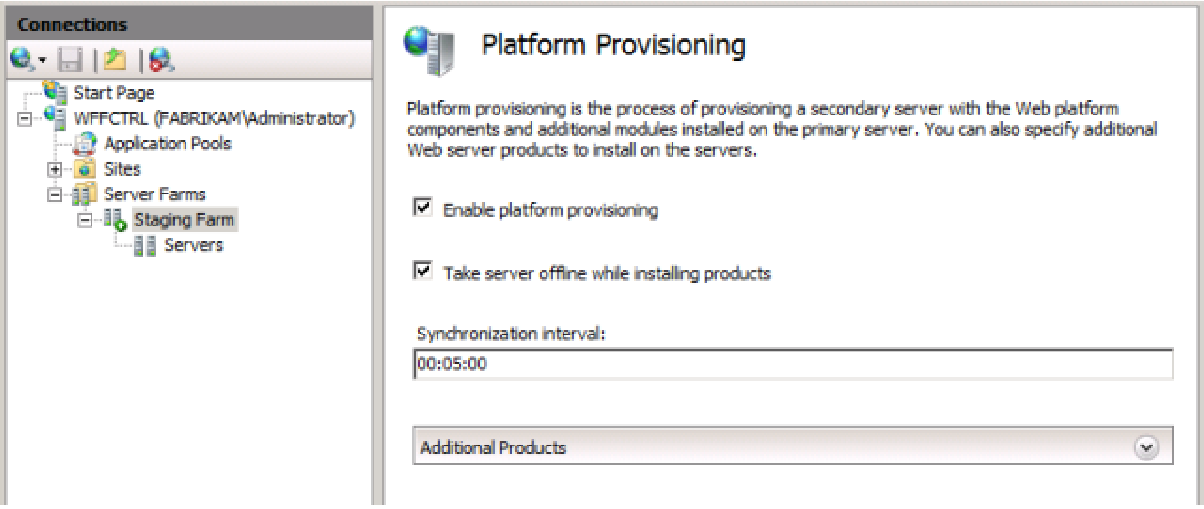
☑ Enable application provisioning

☑ Take server offline while syncing applications

Synchronization interval: 00:00:30

As you can see, the server farm is currently configured to synchronize web content and configuration settings between the primary server and the secondary servers every 30 seconds.

3. Click **Back**, and then double-click **Platform Provisioning**.



Platform Provisioning

Platform provisioning is the process of provisioning a secondary server with the Web platform components and additional modules installed on the primary server. You can also specify additional Web server products to install on the servers.

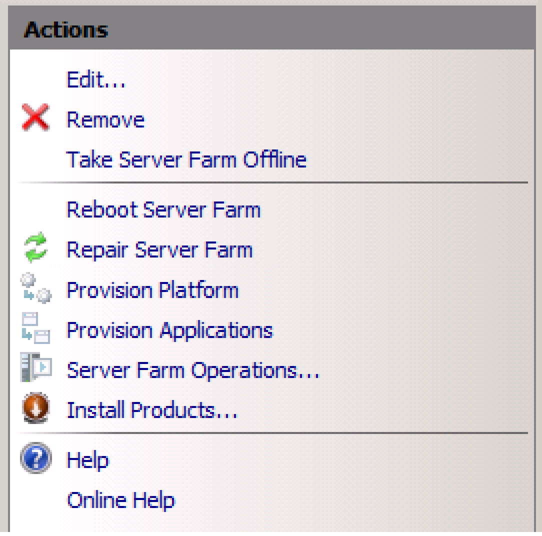
☑ Enable platform provisioning

☑ Take server offline while installing products

Synchronization interval: 00:05:00

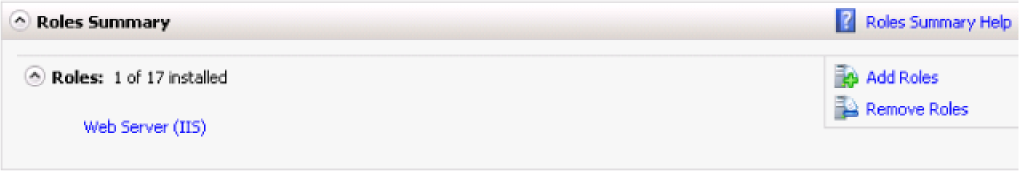
As you can see, the server farm is currently configured to synchronize web platform products and components between the primary server and the secondary servers every five minutes.

4. Click **Back**.
5. To force the server farm to synchronize web platform products immediately, in the **Actions** pane, click **Provision Platform**.



> **Note:** Platform provisioning may take some time. The installer process runs in the background on the secondary servers in your server farm.

6. Once you've allowed sufficient time for the provisioning process to complete, you can verify that the products and components that you added to the primary server have now been replicated on the secondary servers. For example, you can log on to a secondary server and use the **Server Manager** window to verify that the web server role has been installed.

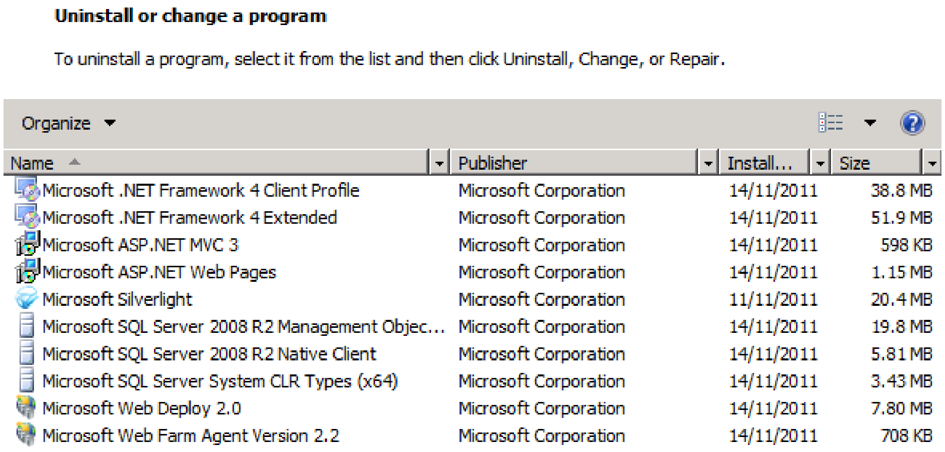


Roles Summary

Roles: 1 of 17 installed

Web Server (IIS)

You can also check the installed programs list to verify that various web platform components have been added.



Uninstall or change a program

To uninstall a program, select it from the list and then click Uninstall, Change, or Repair.

Name	Publisher	Install...	Size
Microsoft .NET Framework 4 Client Profile	Microsoft Corporation	14/11/2011	38.8 MB
Microsoft .NET Framework 4 Extended	Microsoft Corporation	14/11/2011	51.9 MB
Microsoft ASP.NET MVC 3	Microsoft Corporation	14/11/2011	598 KB
Microsoft ASP.NET Web Pages	Microsoft Corporation	14/11/2011	1.15 MB
Microsoft Silverlight	Microsoft Corporation	11/11/2011	20.4 MB
Microsoft SQL Server 2008 R2 Management Objec...	Microsoft Corporation	14/11/2011	19.8 MB
Microsoft SQL Server 2008 R2 Native Client	Microsoft Corporation	14/11/2011	5.81 MB
Microsoft SQL Server System CLR Types (x64)	Microsoft Corporation	14/11/2011	3.43 MB
Microsoft Web Deploy 2.0	Microsoft Corporation	14/11/2011	7.80 MB
Microsoft Web Farm Agent Version 2.2	Microsoft Corporation	14/11/2011	708 KB

Configure Load Balancing

When you create a web farm, you need to set up some form of load balancing to distribute HTTP requests between your web servers. This could be Windows Server 2008 network load balancing, IIS ARR, or a third-party software-based or hardware-based load balancing solution.

WFF is designed to integrate closely with IIS ARR. To take advantage of this integration, you need to install the ARR module on the WFF controller server. You then direct all your web traffic to the controller server, typically by configuring Domain Name System (DNS) records. The controller server will then distribute incoming requests among the servers in your farm, based on server availability and various other criteria.

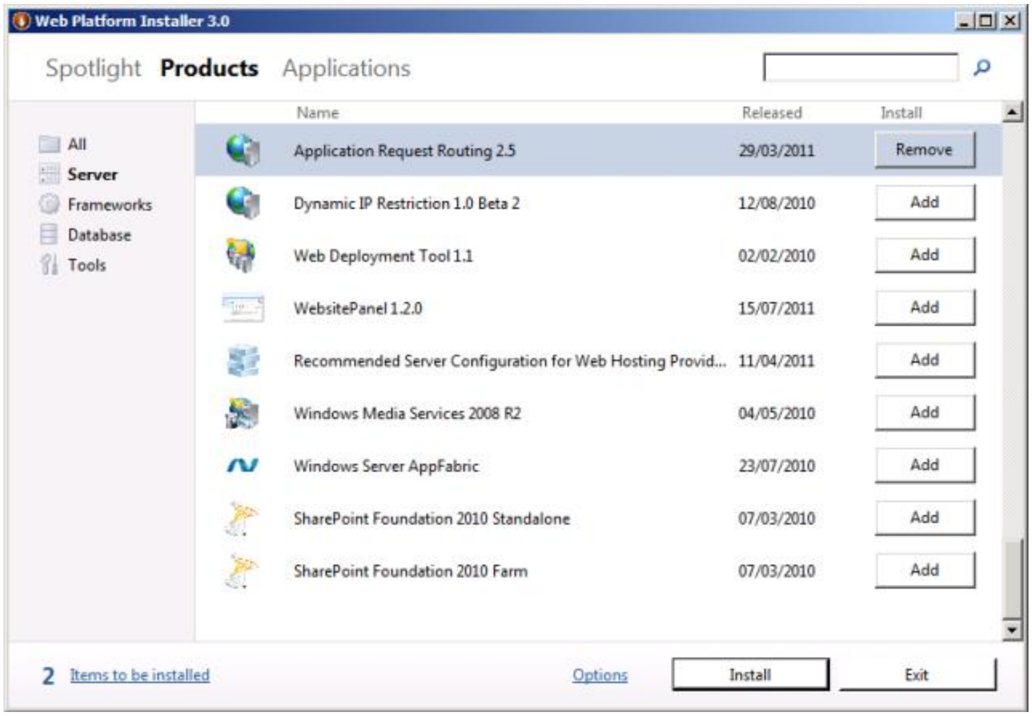
> **Note:** You don't have to use ARR with WFF; you can configure WFF to work with third-party load balancing solutions. For more information, see Overview of the Web Farm Framework 2.0 for IIS 7.

Load balancing using ARR is a complex topic, most of which is beyond the scope of this tutorial. However, you can use the next procedure to install the ARR module and get started with load balancing.

To set up load balancing on the WFF controller server

1. On the WFF controller server, launch the Web Platform Installer.
2. At the top of the **Web Platform Installer 3.0** window, click **Products**.
3. On the left side of the window, in the navigation pane, click **Server**.

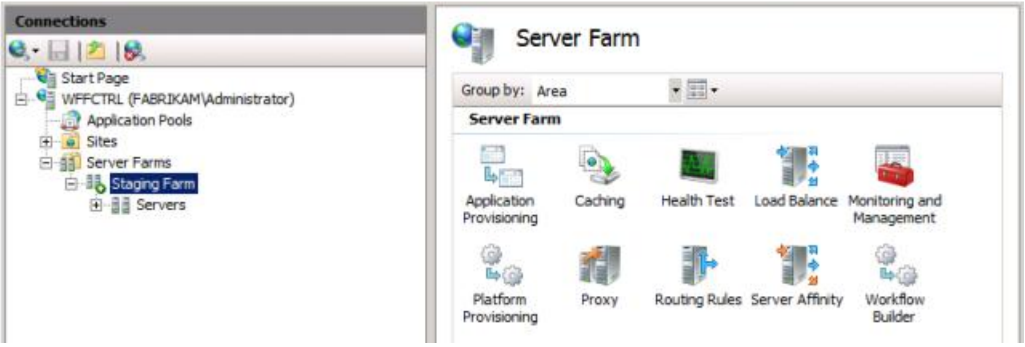
4. In the **Application Request Routing 2.5** row, click **Add**.



Name	Released	Install
Application Request Routing 2.5	29/03/2011	Remove
Dynamic IP Restriction 1.0 Beta 2	12/08/2010	Add
Web Deployment Tool 1.1	02/02/2010	Add
WebsitePanel 1.2.0	15/07/2011	Add
Recommended Server Configuration for Web Hosting Provid...	11/04/2011	Add
Windows Media Services 2008 R2	04/05/2010	Add
Windows Server AppFabric	23/07/2010	Add
SharePoint Foundation 2010 Standalone	07/03/2010	Add
SharePoint Foundation 2010 Farm	07/03/2010	Add

5. Click **Install**, and then follow the instructions in the **Web Platform Installation** window.

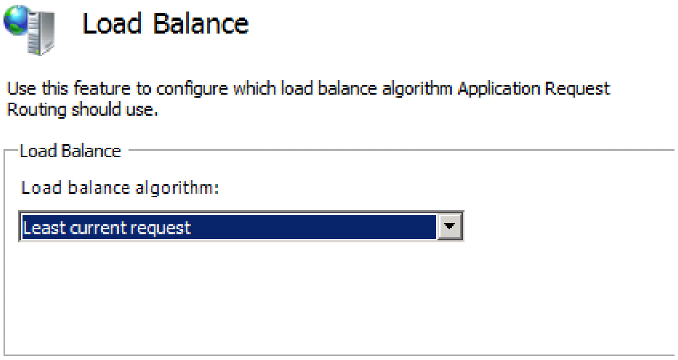
6. When the installation is complete, launch IIS Manager, and in the **Connections** pane, click your server farm node. Notice that several new icons have been added to the **Server Farm** pane.



7. In the **Server Farm** pane, double-click **Load Balance**.

8. In the **Load Balance** pane, select a load balance algorithm (for example, **Least current request**).

> **Note:** For more information on load balancing algorithms and other configuration settings, see Application Request Routing Module.



Load Balance

Use this feature to configure which load balance algorithm Application Request Routing should use.

Load Balance

Load balance algorithm: Least current request

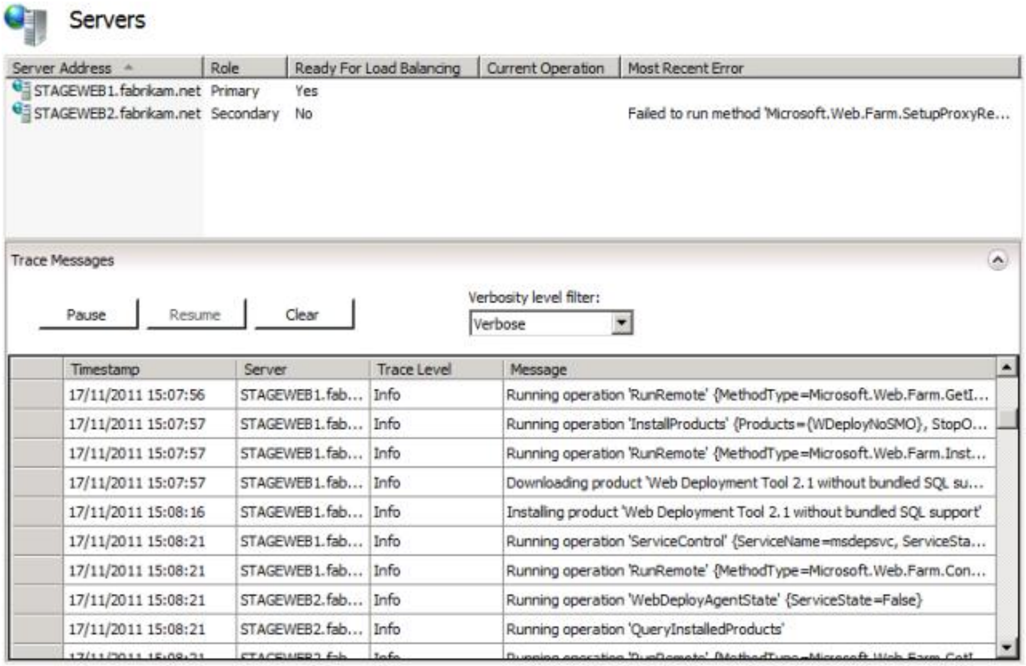
9. In the **Actions** pane, click **Apply**.

You have now configured basic load balancing for the servers in your server farm. If you direct all your web farm traffic to the controller server, the requests will be distributed between the servers in your farm according to availability and the load balancing algorithm you selected.

For more information on how to configure load balancing with ARR, see Application Request Routing Module.

Monitor the Server Farm

You can monitor the health of your server farm at any time through IIS Manager on the controller server. In the **Connections** pane, expand your server farm, and then click **Servers**. The center pane will show a summary of each server in the farm together with a trace log of recent activity.



Servers

Server Address	Role	Ready For Load Balancing	Current Operation	Most Recent Error
STAGEWEB1.fabrikam.net	Primary	Yes		
STAGEWEB2.fabrikam.net	Secondary	No		Failed to run method 'Microsoft.Web.Farm.SetupProxyRe...

Trace Messages

Verbosity level filter: Verbose

Timestamp	Server	Trace Level	Message
17/11/2011 15:07:56	STAGEWEB1.fab...	Info	Running operation 'RunRemote' {MethodType=Microsoft.Web.Farm.GetI...
17/11/2011 15:07:57	STAGEWEB1.fab...	Info	Running operation 'InstallProducts' {Products=(WDeployNoSMO}, StopO...
17/11/2011 15:07:57	STAGEWEB1.fab...	Info	Running operation 'RunRemote' {MethodType=Microsoft.Web.Farm.Inst...
17/11/2011 15:07:57	STAGEWEB1.fab...	Info	Downloading product 'Web Deployment Tool 2.1 without bundled SQL su...
17/11/2011 15:08:16	STAGEWEB1.fab...	Info	Installing product 'Web Deployment Tool 2.1 without bundled SQL support'
17/11/2011 15:08:21	STAGEWEB1.fab...	Info	Running operation 'ServiceControl' {ServiceName=msdepsvc, ServiceSta...
17/11/2011 15:08:21	STAGEWEB1.fab...	Info	Running operation 'RunRemote' {MethodType=Microsoft.Web.Farm.Con...
17/11/2011 15:08:21	STAGEWEB2.fab...	Info	Running operation 'WebDeployAgentState' {ServiceState=False}
17/11/2011 15:08:21	STAGEWEB2.fab...	Info	Running operation 'QueryInstalledProducts'

Conclusion

Your WFF server farm should now be up and running. You can configure the primary server to support whichever deployment approach you prefer—see the Further Reading section for details—and your configuration will be replicated on each secondary server in the server farm.

Further Reading

For more guidance on all aspects of configuring and using the WFF, see the Microsoft Web Farm Framework 2.0 for IIS 7 website.

Configuring Deployment Properties for a Target Environment

This topic describes how to configure environment-specific properties in order to deploy the sample Contact Manager solution to a specific target environment.

Process Overview

The project file that you'll use to build and deploy the Contact Manager solution is split into two physical files:

- One that contains universal build settings and instructions (the *Publish.proj* file).
- One that contains environment-specific build settings (*Env-Dev.proj*, *Env-Stage.proj*, and so on).

At build time, the appropriate environment-specific project file is merged into the universal *Publish.proj* file to form a complete set of build instructions. You can configure deployment to specific destination environments by creating or customizing environment-specific project files with settings that describe your own deployment scenario.

Lots of these values are determined by how your destination environment is configured—in particular, whether your target web server is configured to use the Web Deployment Agent Service (the remote agent) or the Web Deploy Handler. For more information on these approaches, and for guidance on choosing the right approach for your own environment, see Choosing the Right Approach to Web Deployment.

The Contact Manager scenario requires two environment-specific project files:

- Deployment to a developer test environment (*Env-Dev.proj*). The developer test environment is configured to accept remote deployments using the remote agent, as described in Scenario: Configuring a Test Environment for Web Deployment. This file needs to provide the remote agent endpoint address as well as location-specific settings like connection strings and service endpoints.
- Deployment to a staging environment (*Env-Stage.proj*). The staging environment is configured to accept remote deployments using the Web Deploy Handler, as described in Scenario: Configuring a Staging Environment for Web Deployment. This file needs to provide the Web

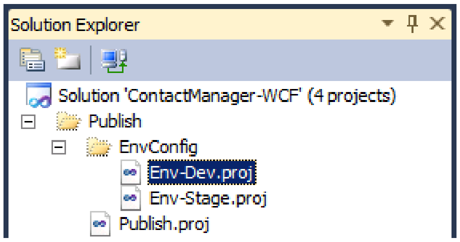
Deploy Handler endpoint address as well as location-specific settings like connection strings and service endpoints.

It's important to note that the settings you configure in the environment-specific project file don't affect the contents of the web package itself—instead, they control how the package is deployed and what parameter values are provided when the package is extracted. You're importing the web package into the production environment manually, as described in Scenario: Configuring a Production Environment for Web Deployment and Manually Installing Web Packages, so it doesn't matter what settings you used in the environment-specific project file when you generated the package. Internet Information Services (IIS) Manager will prompt you for any parameterized values, like connection strings and service endpoints, when you import the package.

To deploy the Contact Manager solution to your own target environment, you can either customize this file or use it as a template and create your own file.

To configure environment-specific deployment settings for the Contact Manager solution

1. Open the ContactManager-WCF solution in Visual Studio 2010.
2. In the **Solution Explorer** window, expand the **Publish** folder, expand the **EnvConfig** folder, and then double-click **Env-Dev.proj**.



3. Replace the property values in the *Env-Dev.proj* file with the correct values for your own test environment.

Note: The table that follows this procedure provides more information on each of these properties.

4. Save your work, and then close the *Env-Dev.proj* file.

Choosing the Right Deployment Properties

This table describes the purpose of each property in the sample environment-specific project file, *Env-Dev.proj*, and provides some guidance on the values you should provide.

Property Name	Details

Property Name	Details
MSDeployComputerName The name of the destination web server or service endpoint.	If you're deploying to the remote agent service on the destination web server, you can specify the target computer name (for example, **TESTWEB1** or **TESTWEB1.fabrikam.net**), or you can specify the remote agent endpoint (for example, **http://TESTWEB1/MSDEPLOYAGENTSERVICE**). The deployment works the same way in each case. If you're deploying to the Web Deploy Handler on the destination web server, you should specify the service endpoint and include the name of the IIS website as a query string parameter (for example, **https://STAGEWEB1:8172/MSDeploy.axd?site=DemoSite**).
MSDeployAuth The method that Web Deploy should use to authenticate to the remote computer.	This should be set to **NTLM** or **Basic**. Typically, you'll use **NTLM** if you're deploying to the remote agent service and **Basic** if you're deploying to the Web Deploy Handler. If you use basic authentication, you also need to specify the user name and password that the IIS Web Deployment Tool (Web Deploy) should impersonate in order to perform the deployment. In this example, these values are provided through the **MSDeployUsername** and **MSDeployPassword** properties. If you use NTLM authentication, you can omit these properties or leave them blank.
MSDeployUsername If you use basic authentication, Web Deploy will use this account on the remote computer.	This should take the form *DOMAIN\username* (for example, **FABRIKAM\matt**). This value is only used if you specify basic authentication. If you use NTLM authentication, the property can be omitted. If a value is supplied, it will be ignored.
MSDeployPassword If you use basic authentication, Web Deploy will use this password on the remote computer.	This is the password for the user account you specified in the **MSDeployUsername** property. This value is only used if you specify basic authentication. If you use NTLM authentication, the property can be omitted. If a value is supplied, it will be ignored.
ContactManagerIisPath The IIS path on which you want to deploy the Contact Manager MVC application.	This should be the path as it appears in IIS Manager, in the form [*IIS website name*]/[*web application name*]. Remember that the IIS website needs to exist before you deploy your application. For example, if you've created an IIS website named DemoSite, you could specify the IIS path for the MVC application as DemoSite/ContactManager.
ContactManagerServiceIisPath The IIS path on which you want to deploy the Contact Manager WCF service.	For example, if you've created an IIS website named DemoSite, you could specify the IIS path for the WCF service as **DemoSite/ContactManagerService**.

Property Name	Details
ContactManagerTargetUrl The URL at which the WCF service can be reached.	This will take the form [*IIS website root URL*]/[*service application name*]/[*service endpoint*]. For example, if you've created an IIS website on port 85, the URL would take the form **http://localhost:85/ContactManagerService/ContactService.svc**. Remember that the MVC application and the WCF service are deployed to the same server. As a result, this URL is only ever accessed from the machine on which it's installed. Because of this, it's better to use localhost or the IP address, rather than the machine name or a host header, in the URL. If you use the machine name or a host header, the loopback check security feature in IIS may block the URL and return an **HTTP 401.1 - Unauthorized** error.
CmDatabaseConnectionString The connection string for the database server.	The connection string determines both the credentials that VSDBCMD will use to contact the database server and create the database and the credentials that the web server application pool will use to contact the database server and interact with the database. Essentially you have two choices here. You can specify **Integrated Security=true**, in which case integrated Windows authentication is used: **Data Source=TESTDB1;Integrated Security=true** In this case, the database will be created using the credentials of the user who runs the VSDBCMD executable, and the application will access the database using the identity of the web server machine account. Alternatively, you can specify the user name and password of a SQL Server account. In this case, the SQL Server credentials are used both by VSDBCMD to create the database and by the application pool to interact with the database: **Data Source=TESTDB1;User Id=ASqlUser; Password=Pa$$w0rd** The walkthroughs in this topic assume that you'll use integrated Windows authentication.
CmTargetDatabase The name you want to give the database you'll create on the database server.	The value you provide here is added to the VSDBCMD command as a parameter. It's also used to build a full connection string that the application pool on the web server can use to interact with the database.

These examples show how you might configure these properties for specific deployment scenarios.

Example 1—Deployment to the Remote Agent Service

In this example:

- You're deploying to the remote agent service on TESTWEB1.
- You're instructing Web Deploy to use NTLM authentication. Web Deploy will run using the credentials you used to invoke the Microsoft Build Engine (MSBuild).

- You're using integrated authentication to deploy the **ContactManager** database to TESTDB1. The database will be deployed using the credentials you used to invoke MSBuild.

XML

```xml
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <MSDeployComputerName Condition=" '$(MSDeployComputerName)'=='' ">
      TESTWEB1.fabrikam.net
    </MSDeployComputerName>
    <MSDeployAuth Condition=" '$(MSDeployAuth)'=='' ">NTLM</MSDeployAuth>
    <ContactManagerTargetUrl Condition =" '$(ContactManagerTargetUrl)'=='' ">
      http://localhost:85/ContactManagerService/ContactService.svc
    </ContactManagerTargetUrl>
    <ContactManagerIisPath Condition=" '$(ContactManagerIisPath)'=='' ">
      DemoSite/ContactManager
    </ContactManagerIisPath>
    <ContactManagerServiceIisPath
      Condition=" '$(ContactManagerServiceIisPath)'=='' ">
        DemoSite/ContactManagerService
    </ContactManagerServiceIisPath>
    <CmDatabaseConnectionString Condition=" '$(CmDatabaseConnectionString)'=='' ">
      Data Source=TESTDB1;Integrated Security=true</CmDatabaseConnectionString>
    <CmTargetDatabase Condition=" '$(CmTargetDatabase)'=='' ">
      ContactManager
    </CmTargetDatabase>
  </PropertyGroup>
</Project>
```

Example 2—Deployment to the Web Deploy Handler Endpoint

In this example:

- You're deploying to the Web Deploy Handler service endpoint on STAGEWEB1.
- You're instructing Web Deploy to use basic authentication.
- You're specifying that Web Deploy should impersonate the FABRIKAM\stagingdeployer account on the remote computer.
- You're using SQL Server authentication to deploy the **ContactManager** database to STAGEDB1.

XML

```xml
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <MSDeployComputerName Condition=" '$(MSDeployComputerName)'=='' ">
      https://STAGEWEB1:8172/MSDeploy.axd?site=DemoSite
    </MSDeployComputerName>
    <MSDeployAuth Condition=" '$(MSDeployAuth)'=='' ">Basic</MSDeployAuth>
    <MSDeployUsername Condition=" '$(MSDeployUsername)'=='' ">
      FABRIKAM\stagingdeployer
    </MSDeployUsername>
    <MSDeployPassword Condition=" '$(MSDeployPassword)'=='' ">
      Pa$$w0rd
    </MSDeployPassword>
    <ContactManagerTargetUrl Condition =" '$(ContactManagerTargetUrl)'=='' ">
      http://localhost:85/ContactManagerService/ContactService.svc
    </ContactManagerTargetUrl>
    <ContactManagerIisPath Condition=" '$(ContactManagerIisPath)'=='' ">
      DemoSite/ContactManager
    </ContactManagerIisPath>
    <ContactManagerServiceIisPath
      Condition=" '$(ContactManagerServiceIisPath)'=='' ">
        DemoSite/ContactManagerService
    </ContactManagerServiceIisPath>
    <CmDatabaseConnectionString Condition=" '$(CmDatabaseConnectionString)'=='' ">
      Data Source=STAGEDB1;User ID=sa;Password=Pa$$w0rd;
    </CmDatabaseConnectionString>
    <CmTargetDatabase Condition=" '$(CmTargetDatabase)'=='' ">
      ContactManager
    </CmTargetDatabase>
  </PropertyGroup>
</Project>
```

Conclusion

At this point, your project files are fully configured to build and deploy the Contact Manager solution to one or more destination environments.

To use these project files as part of a single-step, repeatable deployment process, you need to execute the *Publish.proj* file using MSBuild and pass in the location of the environment-specific project file as a parameter. You can do this in various ways:

- For an overview of MSBuild and an introduction to custom project files, see Understanding the Project File.
- For information on how to formulate an MSBuild command that executes your custom project files, see Deploying Web Packages.
- For information on how to incorporate your MSBuild commands into a command file for single-step, repeatable deployments, see Create and Run a Deployment Command File.
- For information on how to execute your custom project files from Team Build, see Creating a Build Definition that Supports Deployment.

Configuring Team Foundation Server for Web Deployment

This tutorial will show you how to configure Team Foundation Server (TFS) 2010 to build solutions and deploy web content to various target environments. This includes continuous integration (CI) scenarios, where you deploy content automatically every time a developer makes a change. It can also include manual trigger scenarios, where an administrator may want to trigger deployment of a specific build to a staging environment once the build has been verified and validated in the test environment. The topics in this tutorial will guide you through the entire configuration process, including:

- How to create a new team project in TFS.
- How to add content to source control.
- How to configure a build server to support CI and deployment.
- How to create a build definition that includes deployment logic.
- How to configure permissions for automated deployment.

Before You Begin

This tutorial assumes that you have installed TFS 2010 and created a team project collection as part of the initial configuration process. The Team Foundation Installation Guide for Visual Studio 2010 provides comprehensive guidance on these tasks.

Scenario Overview

The high-level scenario for these tutorials is described in Enterprise Web Deployment: Scenario Overview. We recommend that you review this topic before you get started on this tutorial.

How to Use This Tutorial

If this is the first time you've performed the tasks described in this tutorial, or if you want to follow the examples using the sample solution, you should work through the tutorial topics in order. Alternatively, you can use individual topics as guidance for specific tasks. This tutorial includes these topics:

- Creating a Team Project in TFS. A team project is the core unit for source control, process management, and build in TFS. You need to create a team project before you can add content to source control or create build definitions.
- Adding Content to Source Control. Once you've created a team project, you can start adding content to source control. You'll need to add your projects and solutions, together with any external dependencies, before you can start configuring builds.
- Configuring a TFS Build Server for Web Deployment. If you want to build your team project content, you'll need to configure a build server. In most cases, this should be on a separate machine from your TFS installation. To configure a build server, you need to install and configure the TFS build service, install Visual Studio 2010, create build controllers and build

agents, install any products or components that your code needs in order to build successfully, and install the Internet Information Services (IIS) Web Deployment Tool (Web Deploy).

- Creating a Build Definition That Supports Deployment. Before you can start queuing or triggering builds in TFS, you need to create at least one build definition for your team project. The build definition defines every aspect of the build, including which things should be included in the build, what should trigger the build, and where Team Build should send the build outputs. You can configure a build definition to run custom Microsoft Build Engine (MSBuild) project files, which lets you include deployment logic in your automated builds.
- Deploying a Specific Build. In a lot of scenarios, you'll want to deploy a specific build, rather than the latest build, to a target environment. In this case, you can configure a build definition that deploys content from a specific drop folder.
- Configuring Permissions for Team Build Deployment. If the build service is to deploy content as part of an automated build process, you need to grant various permissions to the build service account on any destination web servers and database servers.

Key Technologies

This tutorial focuses on how to use these products and technologies to support automated build and web deployment:

- Visual Studio Team Foundation Server 2010
- Team Build and MSBuild
- Web Deploy

The tutorial also touches on the use of Windows Server 2008 R2, IIS 7.5, SQL Server 2008 R2, ASP.NET 4.0, and ASP.NET MVC 3.

Creating a Team Project in TFS

This topic describes how to create a new team project in Team Foundation Server (TFS) 2010.

Task Overview

To provision and use a new team project in TFS, you'll need to complete these high-level steps:

- Grant permissions to the user who will create the new team project.
- Create the team project.
- Grant permissions to the team members who will work on the project.
- Check in some content.

This topic will show you how to perform these procedures, and it will identify the users and job roles that are likely to be responsible for each procedure. Be aware that, depending on the structure of your organization, each of these tasks may be the responsibility of a different person.

The tasks and walkthroughs in this topic assume that you've installed and configured TFS, and that you've created a team project collection as part of the configuration process. For more information on these assumptions, and for more general background information on the scenario, see Configure a TFS Build Server for Web Deployment.

Grant Permissions to the Team Project Creator

In order to create a new team project, you need these permissions:

- You must have the **Create new projects** permission on the TFS application tier. You typically grant this permission by adding users to the **Project Collection Administrators** TFS group. The **Team Foundation Administrators** global group also includes this permission.
- You must have permission to create new team sites within the SharePoint site collection that corresponds to the TFS team project collection. You typically grant this permission by adding the user to a SharePoint group with **Full Control** rights on the SharePoint site collection.
- If you're using SQL Server Reporting Services features, you must be a member of the **Team Foundation Content Manager** role in Reporting Services.

Who Performs These Procedures?

Typically, the person or group who administers the TFS deployment also performs these procedures.

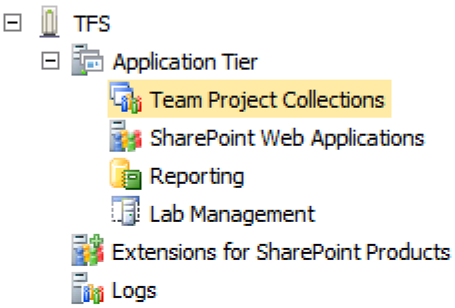
Because this is a highly privileged set of permissions, new team projects are typically created by a small subset of users with responsibility for administering a TFS deployment. Developers will not usually be granted the permissions required to create new team projects.

Grant Permissions in TFS

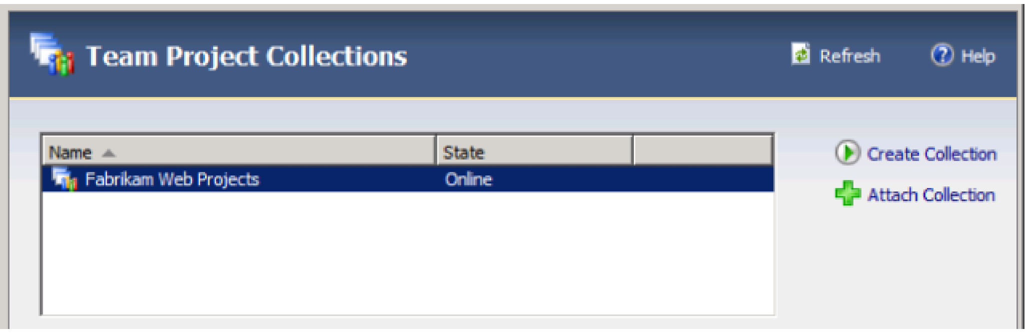
If you want to enable a user to create new team projects, the first high-level task is to add the user to the **Project Collection Administrators** group for the team project collection.

To add a user to the Project Collection Administrators group

1. On the TFS server, on the **Start** menu, point to **All Programs**, click **Microsoft Team Foundation Server 2010**, and then click **Team Foundation Administration Console**.
2. In the navigation tree view, expand **Application Tier**, and then click **Team Project Collections**.

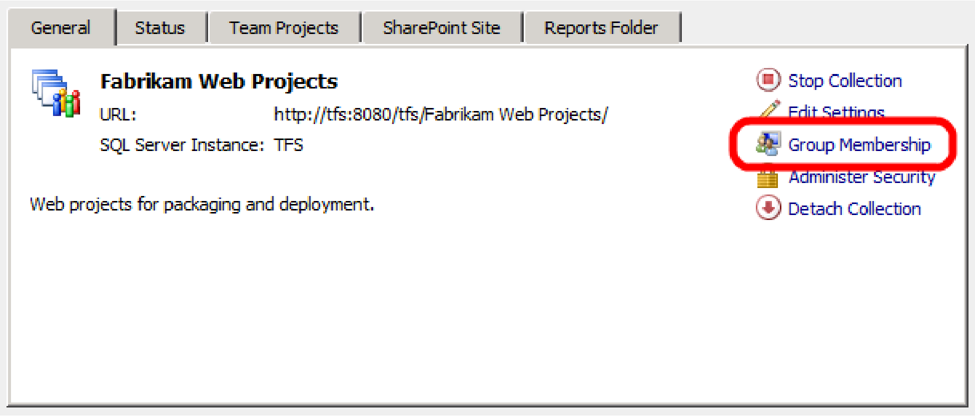


3. In the **Team Project Collections** pane, select the team project collection you want to manage.



Name	State	
Fabrikam Web Projects	Online	

4. On the **General** tab, click **Group Membership**.



Fabrikam Web Projects

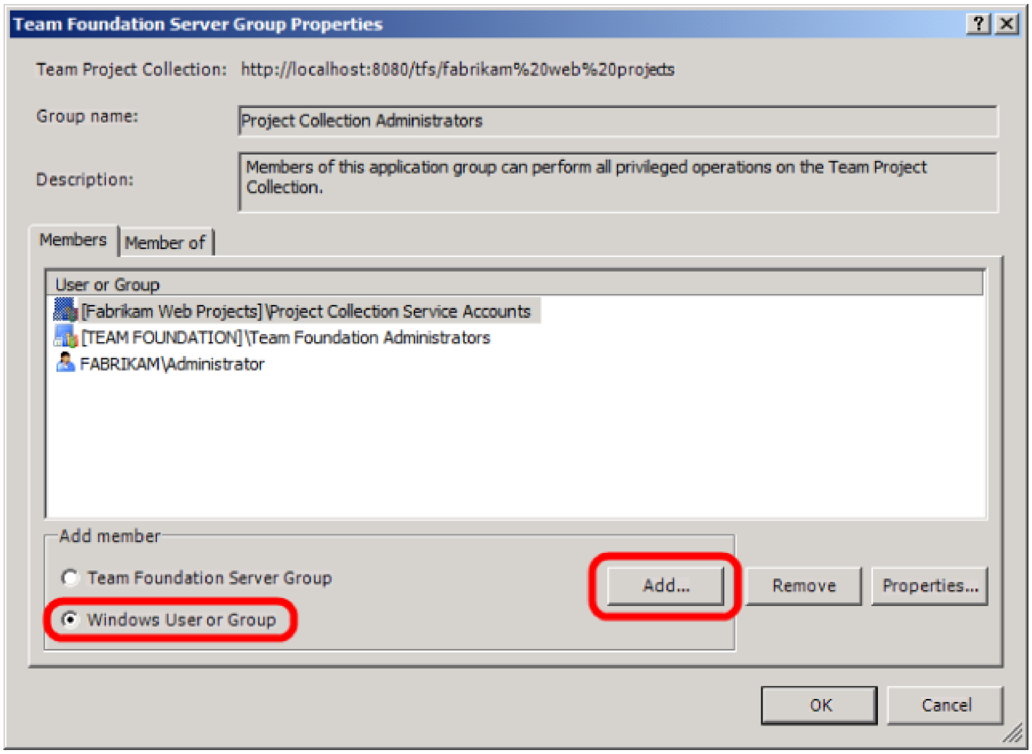
URL: http://tfs:8080/tfs/Fabrikam Web Projects/

SQL Server Instance: TFS

Web projects for packaging and deployment.

5. In the **Global Groups** dialog box, select the **Project Collection Administrators** group, and then click **Properties**.

6. In the **Team Foundation Server Group Properties** dialog box, select **Windows User or Group**, and then click **Add**.



Team Project Collection: http://localhost:8080/tfs/fabrikam%20web%20projects

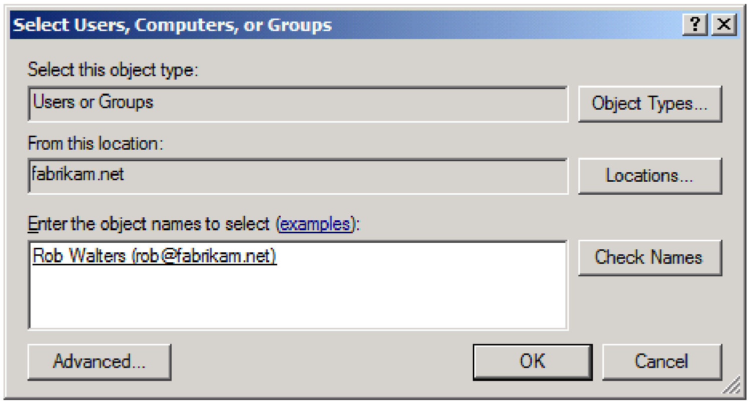
Group name: Project Collection Administrators

Description: Members of this application group can perform all privileged operations on the Team Project Collection.

User or Group
[Fabrikam Web Projects]\Project Collection Service Accounts
[TEAM FOUNDATION]\Team Foundation Administrators
FABRIKAM\Administrator

Add member
○ Team Foundation Server Group
● Windows User or Group

7. In the **Select Users, Computers, or Groups** dialog box, type the user name of the user you want to be able to create new team projects, click **Check Names**, and then click **OK**.



Select this object type: Users or Groups

From this location: fabrikam.net

Enter the object names to select (examples): Rob Walters (rob@fabrikam.net)

8. In the **Team Foundation Server Group Properties** dialog box, click **OK**.

9. In the **Global Groups** dialog box, click **Close**.

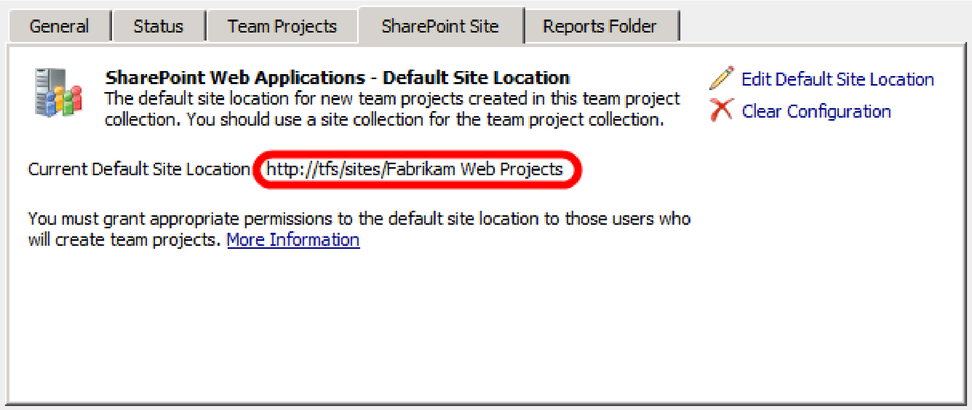
Grant Permissions in SharePoint Services

Next, you need to give the user permission to create new team sites in the SharePoint site collection that corresponds to your TFS team project collection.

To grant Full Control permissions on the SharePoint site collection

1. In the Team Foundation Server Administration Console, on the **Team Project Collections** page, select the team project collection you want to manage.

2. On the **SharePoint Site** tab, note the value of the **Current Default Site Location** URL.



SharePoint Web Applications - Default Site Location
The default site location for new team projects created in this team project collection. You should use a site collection for the team project collection.

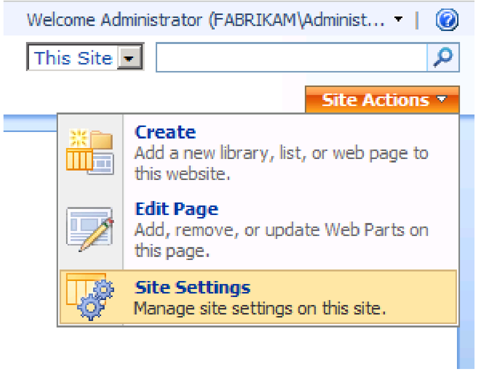
Current Default Site Location http://tfs/sites/Fabrikam Web Projects

You must grant appropriate permissions to the default site location to those users who will create team projects. More Information

3. Open Internet Explorer, and then go to the URL you noted in step 2.

Note: If you're not logged on to Windows as the user who created the team project collection, you'll need to sign in to SharePoint as this user in order to continue.

4. On the **Site Actions** menu, click **Site Settings**.



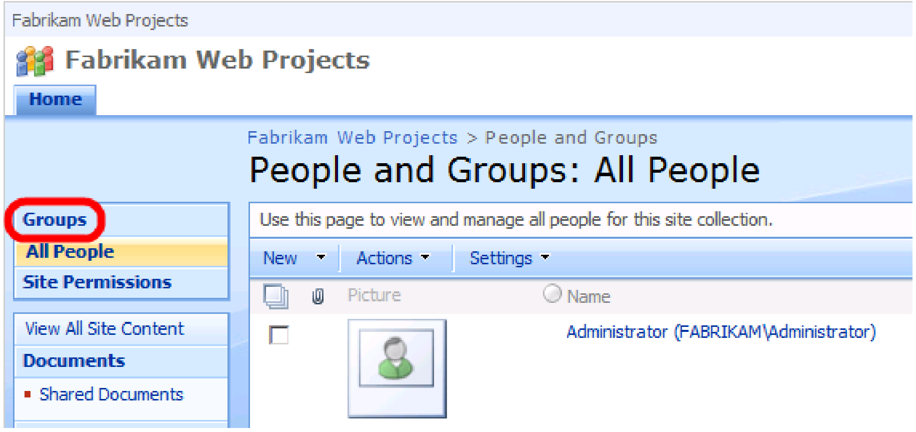
Create
Add a new library, list, or web page to this website.

Edit Page
Add, remove, or update Web Parts on this page.

Site Settings
Manage site settings on this site.

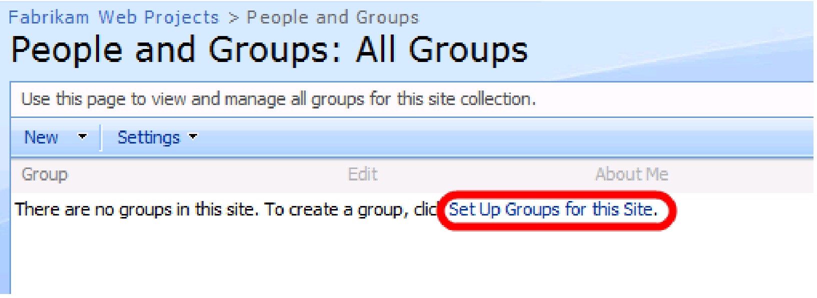
5. On the **Site Settings** page, under **Users and Permissions**, click **People and groups**.

6. In the left navigation panel, click **Groups**.



Use this page to view and manage all people for this site collection.

7. On the **People and Groups: All Groups** page, click **Set Up Groups for this Site**.


Fabrikam Web Projects > People and Groups
People and Groups: All Groups
New
Settings
Group
Edit
About Me
Set Up Groups for this Site.

Use this page to view and manage all groups for this site collection.

There are no groups in this site. To create a group, clic Set Up Groups for this Site.

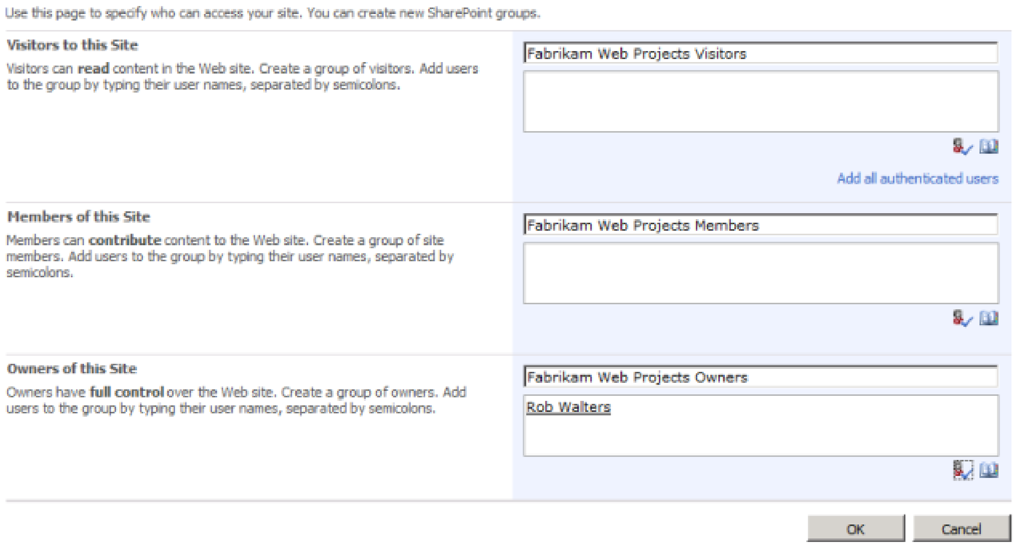
Note: You may receive an **HTTP 404 Not Found** error due to a double HTTP encoding bug. If this occurs, replace the URL with this:

[*site collection URL*]/_layouts/permsetup.aspx

For example:

http://tfs/sites/Fabrikam%20Web%20Projects/_layouts/permsetup.aspx

8. On the **Set Up Groups for this Site** page, add the user who will create team projects to the **Owners** group, and then click **OK**.



Use this page to specify who can access your site. You can create new SharePoint groups.

Visitors to this Site
Visitors can **read** content in the Web site. Create a group of visitors. Add users to the group by typing their user names, separated by semicolons.

Fabrikam Web Projects Visitors

Members of this Site
Members can **contribute** content to the Web site. Create a group of site members. Add users to the group by typing their user names, separated by semicolons.

Fabrikam Web Projects Members

Owners of this Site
Owners have **full control** over the Web site. Create a group of owners. Add users to the group by typing their user names, separated by semicolons.

Fabrikam Web Projects Owners

Rob Walters

For more information on enabling users to create new team projects within a team project collection, see Set Administrator Permissions for Team Project Collections.

Create a New Team Project and Add Users

Once you have the necessary permissions, you can use the **Team Explorer** window in Visual Studio 2010 to create a new team project. This approach provides a wizard that collects all the required information and performs the necessary tasks in TFS, SharePoint, and SQL Server Reporting Services. You'll also need to grant permissions on the new team project to members of the developer team, to enable them to add and modify content.

Who Performs These Procedures?

Usually either a TFS administrator or a developer team leader performs these procedures.

Create a New Team Project

The next procedure describes how to create a new team project in TFS 2010.

To create a new team project

1. On the **Start** menu, point to **All Programs**, click **Microsoft Visual Studio 2010**, right-click **Microsoft Visual Studio 2010**, and then click **Run as administrator**.

   > **Note:** If you don't run Visual Studio 2010 as an administrator, the New Team Project Wizard will fail on the last step.

2. If the **User Account Control** dialog box appears, click **Yes**.

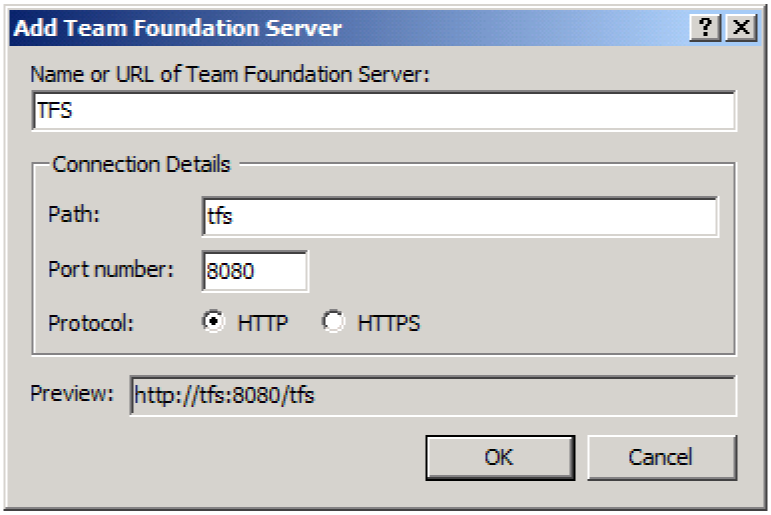
3. In Visual Studio, on the **Team** menu, click **Connect to Team Foundation Server**.

   > **Note:** If you have already configured a connection to a TFS server, you can omit steps 4-7.

4. In the **Connection to Team Project** dialog box, click **Servers**.

5. In the **Add/Remove Team Foundation Server** dialog box, click **Add**.

6. In the **Add Team Foundation Server** dialog box, provide the details of your TFS instance, and then click **OK**.



Name or URL of Team Foundation Server: TFS

Path: tfs

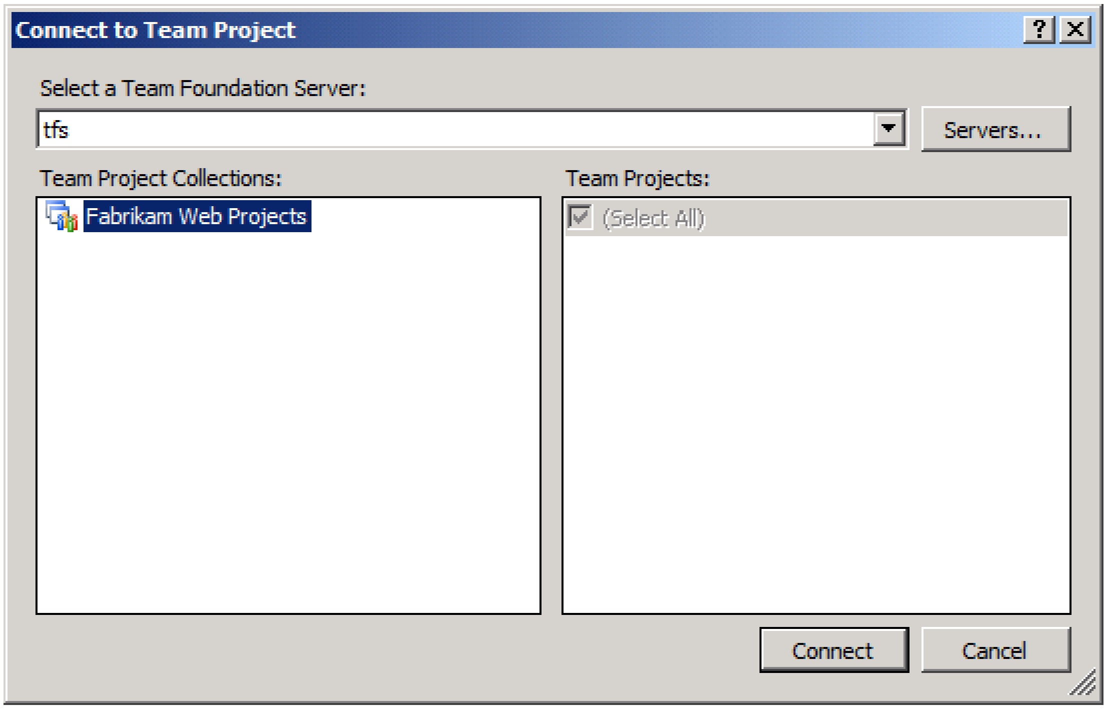
Port number: 8080

Protocol: ● HTTP ○ HTTPS

Preview: http://tfs:8080/tfs

7. In the **Add/Remove Team Foundation Server** dialog box, click **Close**.

8. In the **Connect to Team Project** dialog box, select the TFS instance you want to connect to, select the team project collection you want to add to, and then click **Connect**.

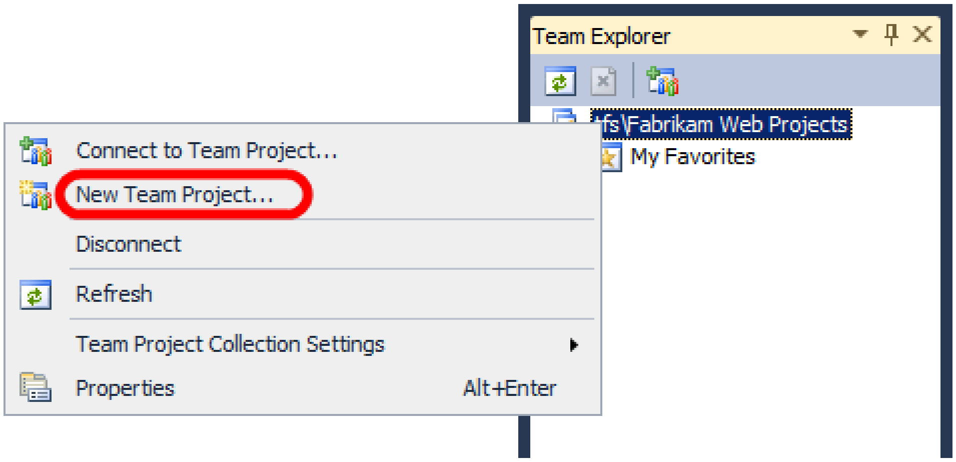


Select a Team Foundation Server: tfs

Team Project Collections: Fabrikam Web Projects

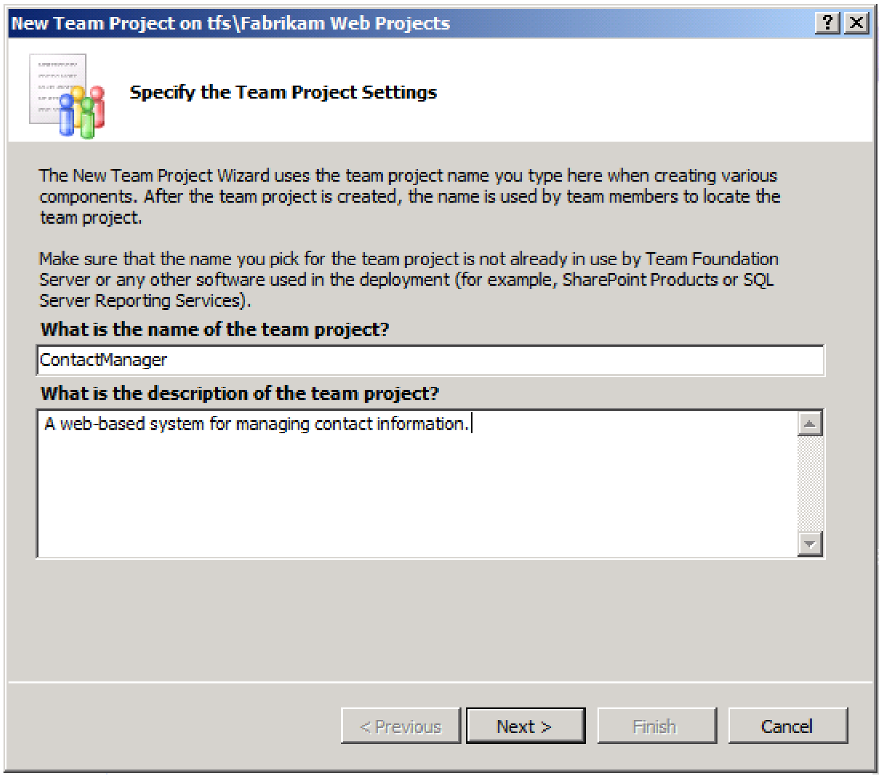
Team Projects: ☑ (Select All)

9. In the **Team Explorer** window, right-click the team project collection, and then click **New Team Project**.


Team Explorer
tfs\Fabrikam Web Projects
My Favorites
Connect to Team Project...
New Team Project...
Disconnect
Refresh
Team Project Collection Settings
Properties
Alt+Enter

10. In the **New Team Project** dialog box, provide a name and a description for the team project, and then click **Next**.

> **Note:** If your team project includes spaces, you may face some issues when you come to use the Internet Information Services (IIS) Web Deployment Tool (Web Deploy) to deploy packages from the output path. Spaces in the path can make it a lot more difficult to run Web Deploy commands.



New Team Project on tfs\Fabrikam Web Projects

Specify the Team Project Settings

The New Team Project Wizard uses the team project name you type here when creating various components. After the team project is created, the name is used by team members to locate the team project.

Make sure that the name you pick for the team project is not already in use by Team Foundation Server or any other software used in the deployment (for example, SharePoint Products or SQL Server Reporting Services).

What is the name of the team project?

ContactManager

What is the description of the team project?

A web-based system for managing contact information.

11. On the **Select a Process Template** page, select the process template that you want to use to manage the development process, and then click **Next**.

> **Note:** For more information on process templates for TFS, see Process Templates and Tools.

12. On the **Team Site Settings** page, leave the default settings unchanged, and then click **Next**.

    This setting creates, or identifies, a SharePoint team site that is associated with the TFS team project. Your development team can use this site to manage documentation, participate in discussion threads, create wiki pages, and perform various other tasks that are not related to code. For more information, see Interactions Between SharePoint Products and Team Foundation Server.

13. On the **Specify Source Control Settings** page, leave the default settings unchanged, and then click **Next**.

    This setting identifies or creates the location in the TFS folder hierarchy that will act as a root folder for your content.

14. On the **Confirm Team Project Settings** page, click **Finish**.

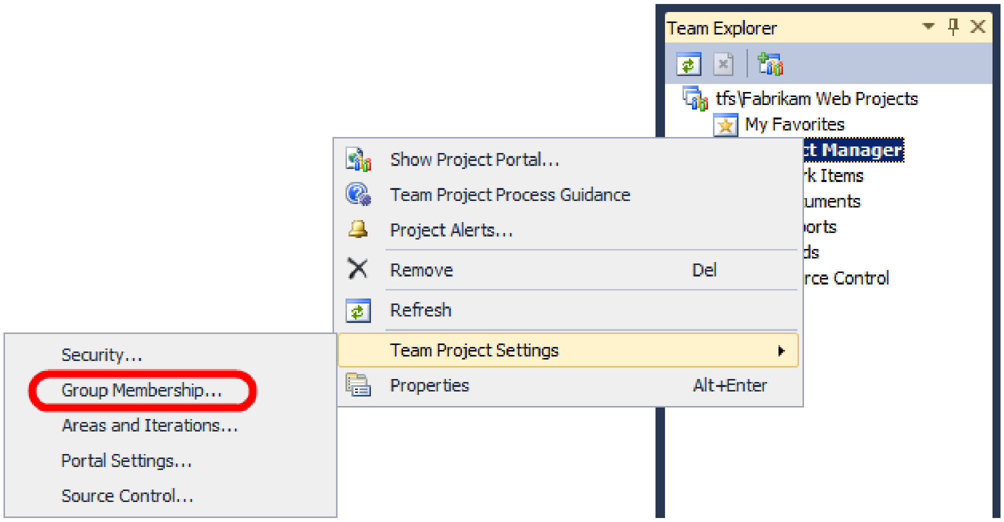
15. When the new team project is successfully created, on the **Team Project Created** page, click **Close**.

Add Users to a Team Project

Now that you've created the new team project, you can grant permissions to users to enable them to start adding and collaborating on content.

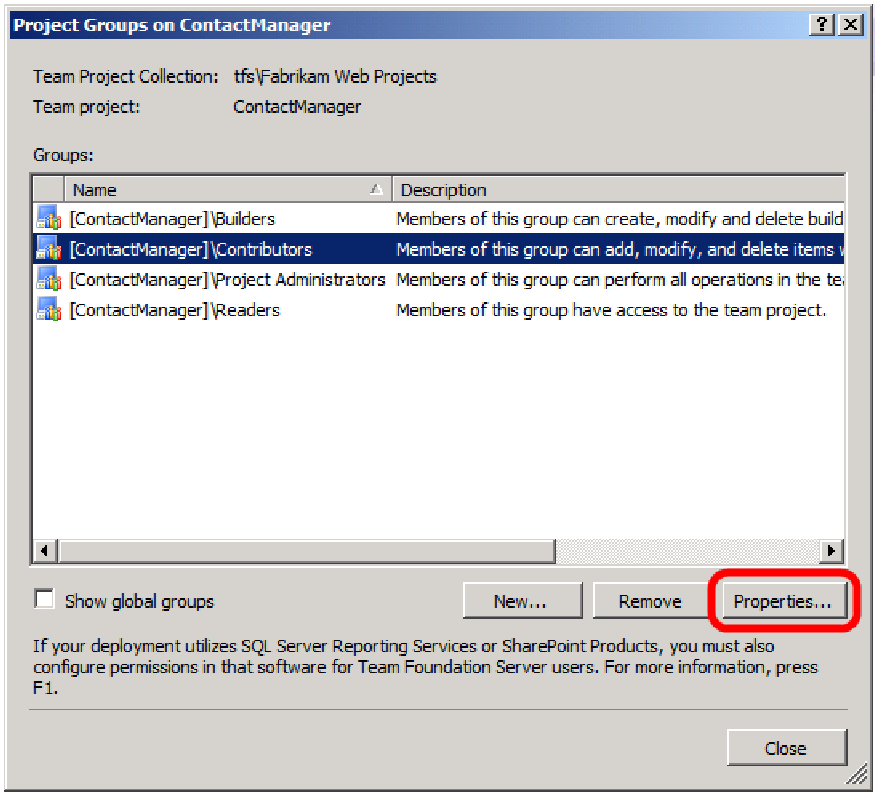
To add users to a team project

1. In Visual Studio 2010, in the **Team Explorer** window, right-click the team project, point to **Team Project Settings**, and then click **Group Membership**.



To enable a user to add, modify, and remove code under source control, you need to add him or her to the **Contributors** group.

2. In the **Project Groups** dialog box, select the **Contributors** group, and then click **Properties**.



Project Groups on ContactManager

Team Project Collection: tfs\Fabrikam Web Projects

Team project: ContactManager

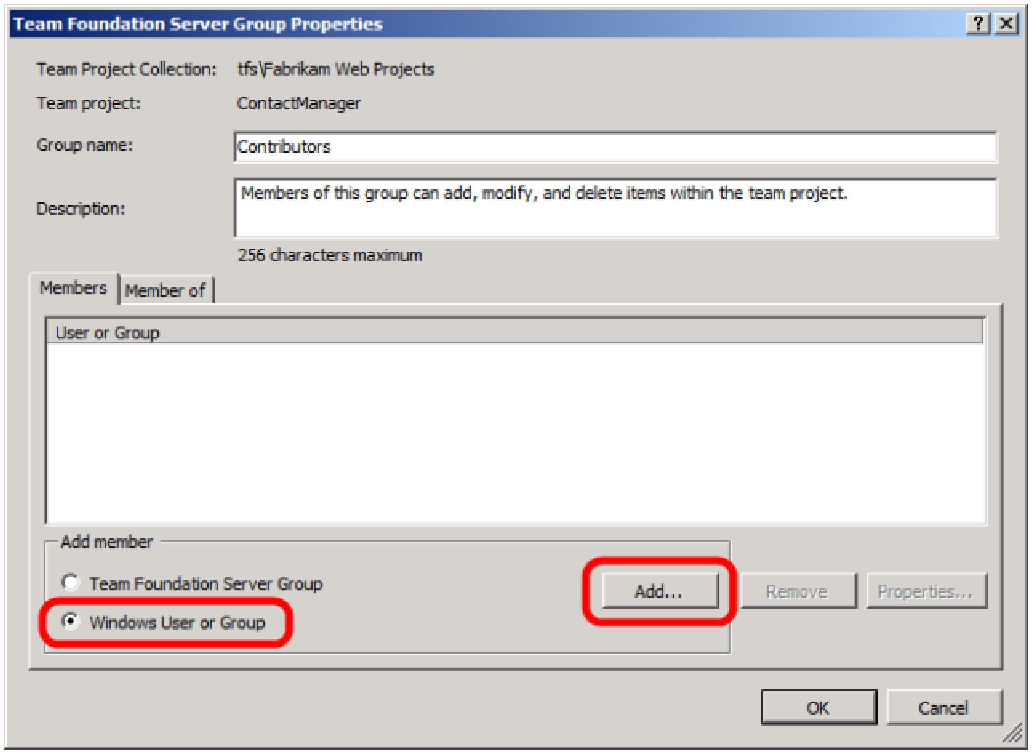
Groups:

Name	Description
[ContactManager]\Builders	Members of this group can create, modify and delete build
[ContactManager]\Contributors	Members of this group can add, modify, and delete items w
[ContactManager]\Project Administrators	Members of this group can perform all operations in the te
[ContactManager]\Readers	Members of this group have access to the team project.

□ Show global groups

If your deployment utilizes SQL Server Reporting Services or SharePoint Products, you must also configure permissions in that software for Team Foundation Server users. For more information, press F1.

3. In the **Team Foundation Server Group Properties** dialog box, select **Windows User or Group**, and then click **Add**.



Team Project Collection: tfs\Fabrikam Web Projects

Team project: ContactManager

Group name: Contributors

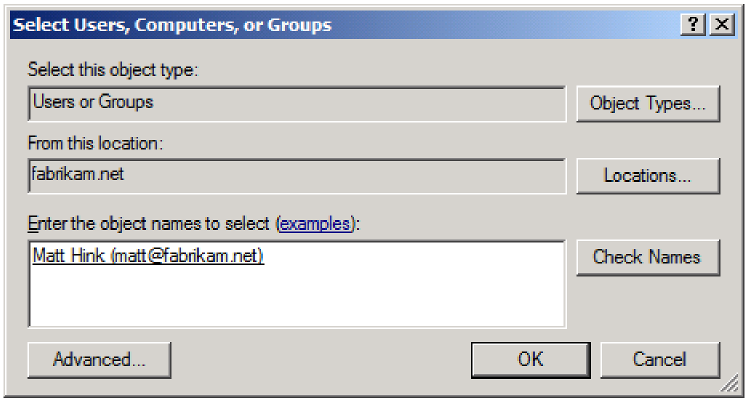
Description: Members of this group can add, modify, and delete items within the team project.

256 characters maximum

○ Team Foundation Server Group

● Windows User or Group

4. In the **Select Users, Computers, or Groups** dialog box, type the user name of the user you want to add to the team project, click **Check Names**, and then click **OK**.



Select this object type: Users or Groups

From this location: fabrikam.net

Enter the object names to select (examples): Matt Hink (matt@fabrikam.net)

5. In the **Team Foundation Server Group Properties** dialog box, click **OK**.
6. In the **Project Groups** dialog box, click **Close**.

Conclusion

At this point, your new team project is ready to use, and your developer team can start adding content and collaborating on the development process.

The next topic, Adding Content to Source Control, describes how to add content to source control.

Further Reading

For broader guidance on creating team projects in TFS, see Create a Team Project. For more information on enabling users to create new team projects within a team project collection, see Set Administrator Permissions for Team Project Collections. For more information on adding users to team projects, see Add Users to Team Projects.

Adding Content to Source Control

This topic explains how to add content to source control in Team Foundation Server (TFS) 2010. It describes how to add solutions and projects to a team project in TFS, and it explains how to add external dependencies like frameworks or assemblies to source control.

Task Overview

In most cases, every member of the developer team should be able to add content to source control. To add a solution to source control in TFS, you'll need to complete these high-level steps:

- Connect to a team project.
- Map the team project folder structure on the server to a folder structure on your local computer.
- Add the solution and its contents to source control.
- Add any external dependencies to source control.

This topic will show you how to perform these procedures.

The tasks and walkthroughs in this topic assume that you've already created a new TFS team project to manage your content. For more information on creating a new team project, see Creating a Team Project in TFS.

Who Performs These Procedures?

In most cases, every member of the developer team should be able to add and modify content within specific team projects.

Connect to a Team Project and Create a Folder Mapping

Before you add any content to source control, you need to connect to a team project and create a mapping between the folder structure on the server and the file system on your local machine.

To connect to a team project and map a local path

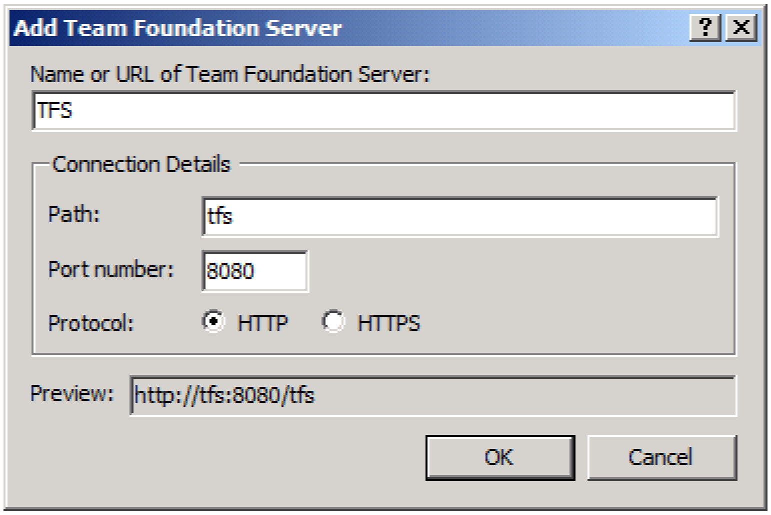
1. On your developer workstation, open Visual Studio 2010.
2. In Visual Studio, on the **Team** menu, click **Connect to Team Foundation Server**.

> **Note:** If you have already configured a connection to a TFS server, you can omit steps 3-6.

3. In the **Connection to Team Project** dialog box, click **Servers**.

4. In the **Add/Remove Team Foundation Server** dialog box, click **Add**.

5. In the **Add Team Foundation Server** dialog box, provide the details of your TFS instance, and then click **OK**.



Name or URL of Team Foundation Server: TFS

Path: tfs

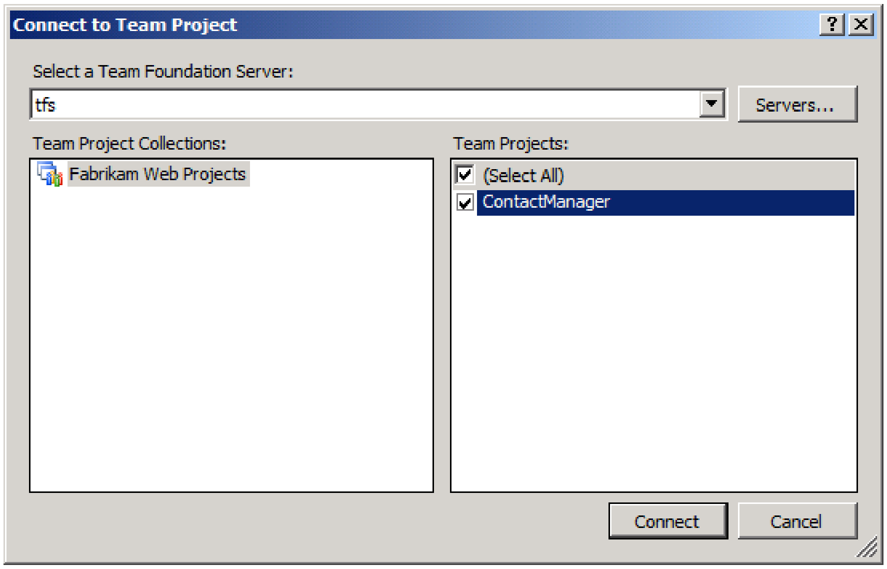
Port number: 8080

Protocol: ● HTTP ○ HTTPS

Preview: http://tfs:8080/tfs

6. In the **Add/Remove Team Foundation Server** dialog box, click **Close**.

7. In the **Connect to Team Project** dialog box, select the TFS instance you want to connect to, select the team project collection, select the team project you want to add to, and then click **Connect**.

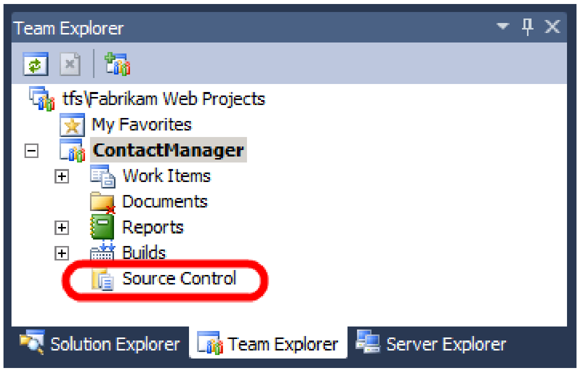


Select a Team Foundation Server: tfs

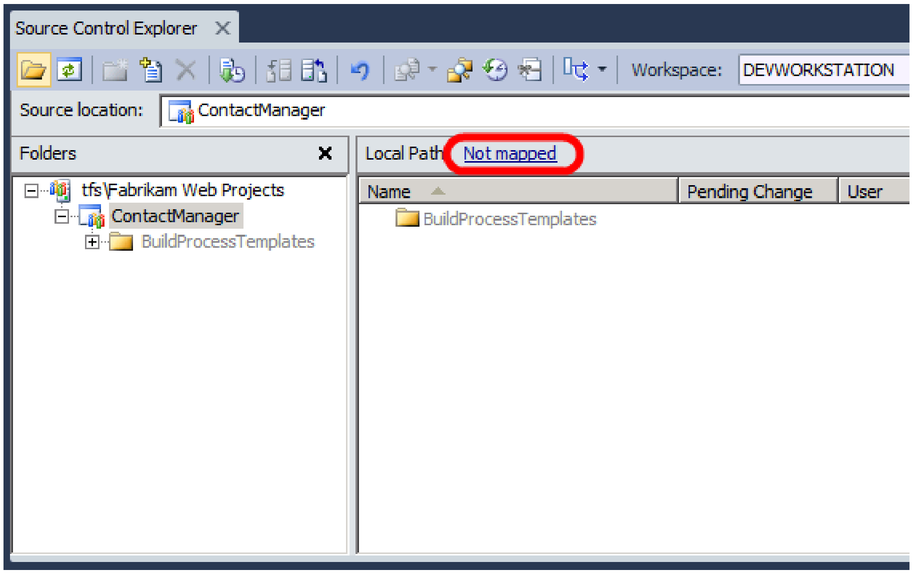
Team Project Collections: Fabrikam Web Projects

Team Projects:
☑ (Select All)
☑ ContactManager

8. In the **Team Explorer** window, expand your team project, and then double-click **Source Control**.



9. On the **Source Control Explorer** tab, click **Not mapped**.

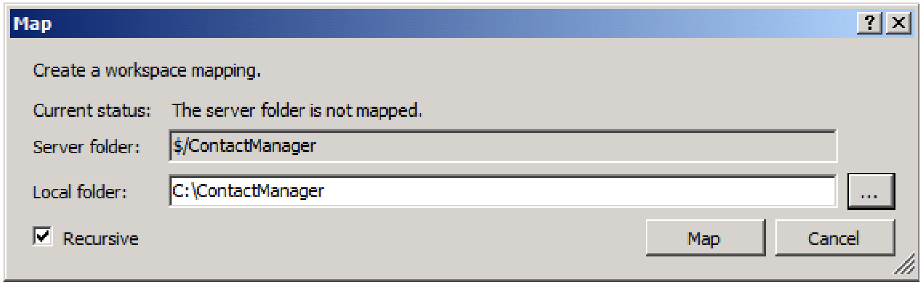


Workspace: DEVWORKSTATION

Source location: ContactManager

Local Path: Not mapped

10. In the **Map** dialog box, in the **Local folder** box, browse to (or create) a local folder to act as the root folder for the team project, and then click **Map**.



Create a workspace mapping.

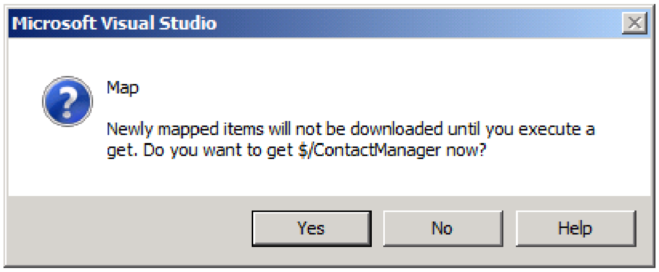
Current status: The server folder is not mapped.

Server folder: $/ContactManager

Local folder: C:\ContactManager

☑ Recursive

11. When you're prompted to download source files, click **Yes**.


Microsoft Visual Studio

Map

Newly mapped items will not be downloaded until you execute a get. Do you want to get $/ContactManager now?

At this point, you have mapped the server-side folder for the team project to a local folder on your developer workstation. You've also downloaded any existing content from the team project to your local folder structure. You can now start to add your own content to source control.

Add Projects and Solutions to Source Control

To add projects and solutions to source control, you first need to move them to the mapped folder for the team project on your local machine. You can then check in the content to synchronize your additions with the server.

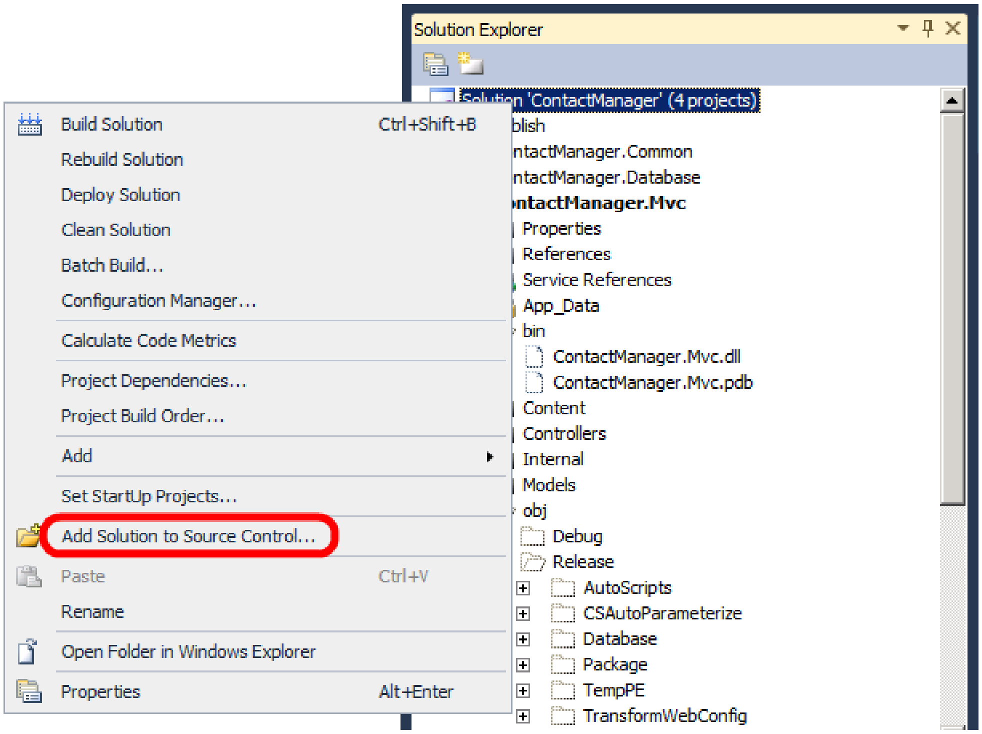
To add projects to source control

1. On your developer workstation, move your projects and solutions to an appropriate location within the mapped folder structure for the team project.

   > **Note:** Many organizations will have a preferred approach to how projects and solutions should be organized in source control. For guidance on how to structure folders, see How To: Structure Your Source Control Folders in Team Foundation Server.

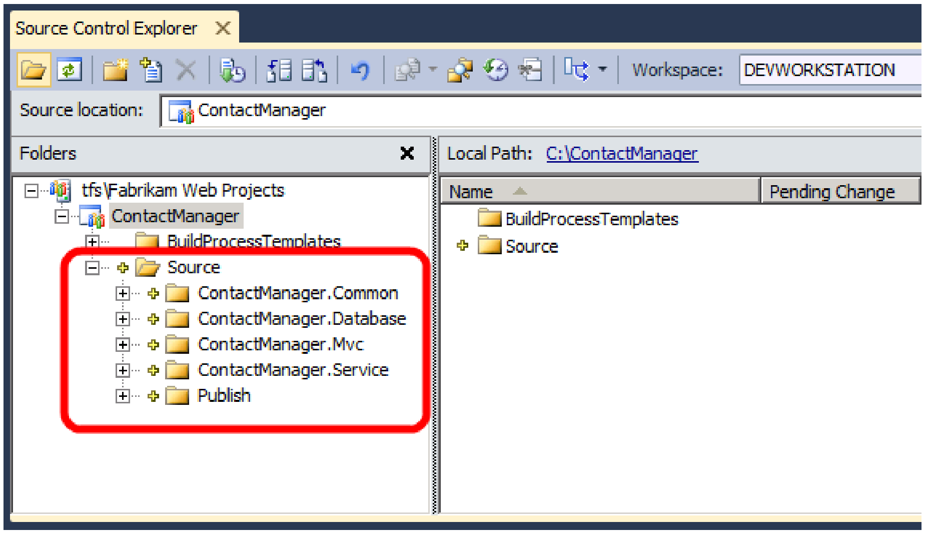
2. Open the solution in Visual Studio 2010.

3. In the **Solution Explorer** window, right-click the solution, and then click **Add Solution to Source Control**.



> **Note:** In some cases, depending on how your organization likes to structure content in TFS, you may need to add projects to source control individually to provide more fine-grained control over how your source code is organized.

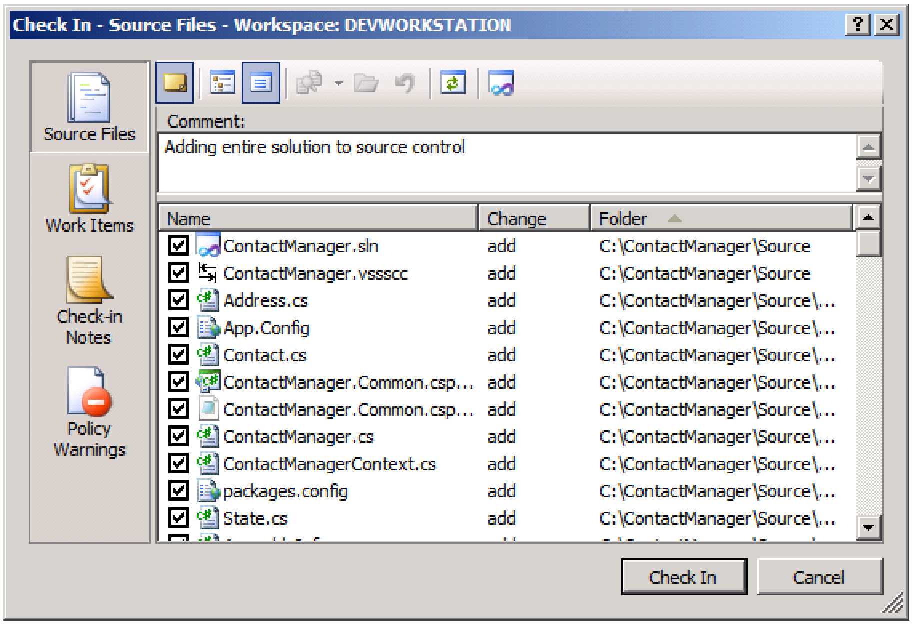
4. Verify that the **Source Control Explorer** tab displays the content you've added within the server folder structure for the team project.



> **Note:** The **Source Control Explorer** tab displays your content with no further prompting because you added your solution to a mapped folder on the local file system. If your solution

was in an unmapped location, you'd be prompted to specify folder locations in both TFS and your local file system.

5. On the **Source Control Explorer** tab, in the **Folders** pane, right-click the team project (for example, **ContactManager**), and then click **Check In Pending Changes**.
6. In the **Check In – Source Files** dialog box, type a comment, and then click **Check In**.



Comment:
Adding entire solution to source control

	Name	Change	Folder
☑	ContactManager.sln	add	C:\ContactManager\Source
☑	ContactManager.vssscc	add	C:\ContactManager\Source
☑	Address.cs	add	C:\ContactManager\Source\...
☑	App.Config	add	C:\ContactManager\Source\...
☑	Contact.cs	add	C:\ContactManager\Source\...
☑	ContactManager.Common.csp...	add	C:\ContactManager\Source\...
☑	ContactManager.Common.csp...	add	C:\ContactManager\Source\...
☑	ContactManager.cs	add	C:\ContactManager\Source\...
☑	ContactManagerContext.cs	add	C:\ContactManager\Source\...
☑	packages.config	add	C:\ContactManager\Source\...
☑	State.cs	add	C:\ContactManager\Source\...

At this point you have added your solution to source control in TFS.

Add External Dependencies to Source Control

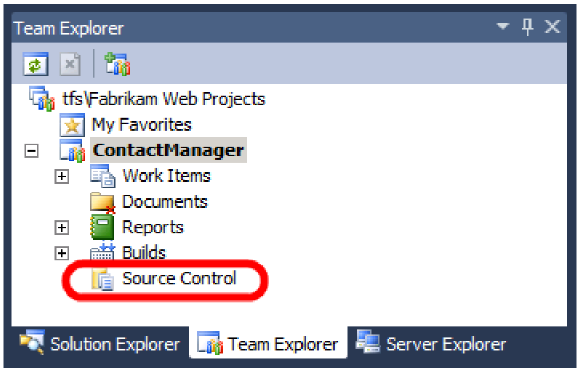
When you add a project or solution to source control, any files and folders within your project or solution will also be added. However, in a lot of cases, projects and solutions also rely on external dependencies, like local assemblies, to function properly. You need to add any such resources to source control to let both Team Build and other members of the developer team build your code successfully.

For example, the folder structure for the Contact Manager sample solution includes a folder named packages. This contains the assembly and various supporting resources for the ADO.NET Entity Framework 4.1. The packages folder is not part of the Contact Manager solution, but the solution will not build successfully without it. To enable Team Build to build the solution, you need to add the packages folder to source control.

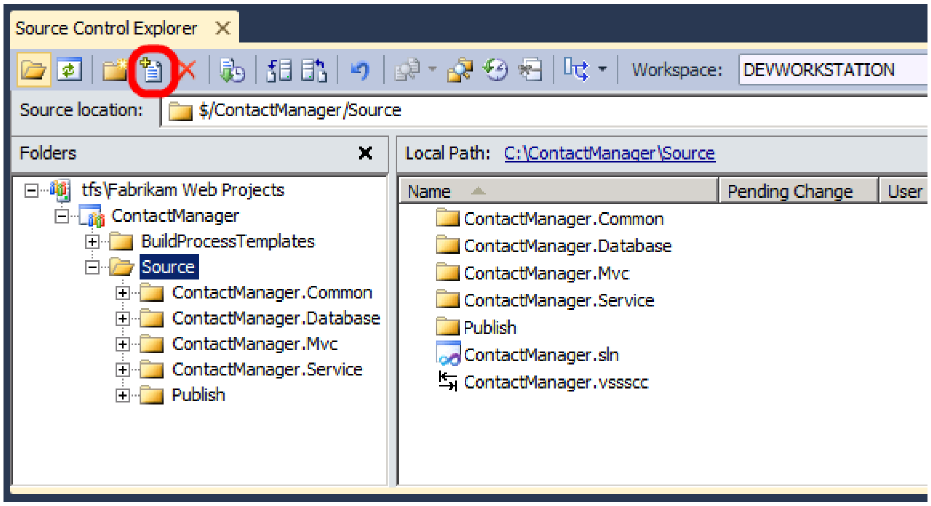
Note: The inclusion of a packages folder is typical of what happens when you add the Entity Framework, or similar resources, to your solution using the NuGet extension for Visual Studio 2010.

To add non-project content to source control

1. Ensure that the items you want to add (for example, the packages folder) are in an appropriate location within a mapped folder on your local file system.
2. In Visual Studio 2010, In the **Team Explorer** window, expand your team project, and then double-click **Source Control**.



3. On the **Source Control Explorer** tab, in the **Folders** pane, select the folder that contains the item or items you want to add.
4. Click the **Add Items to Folder** button.

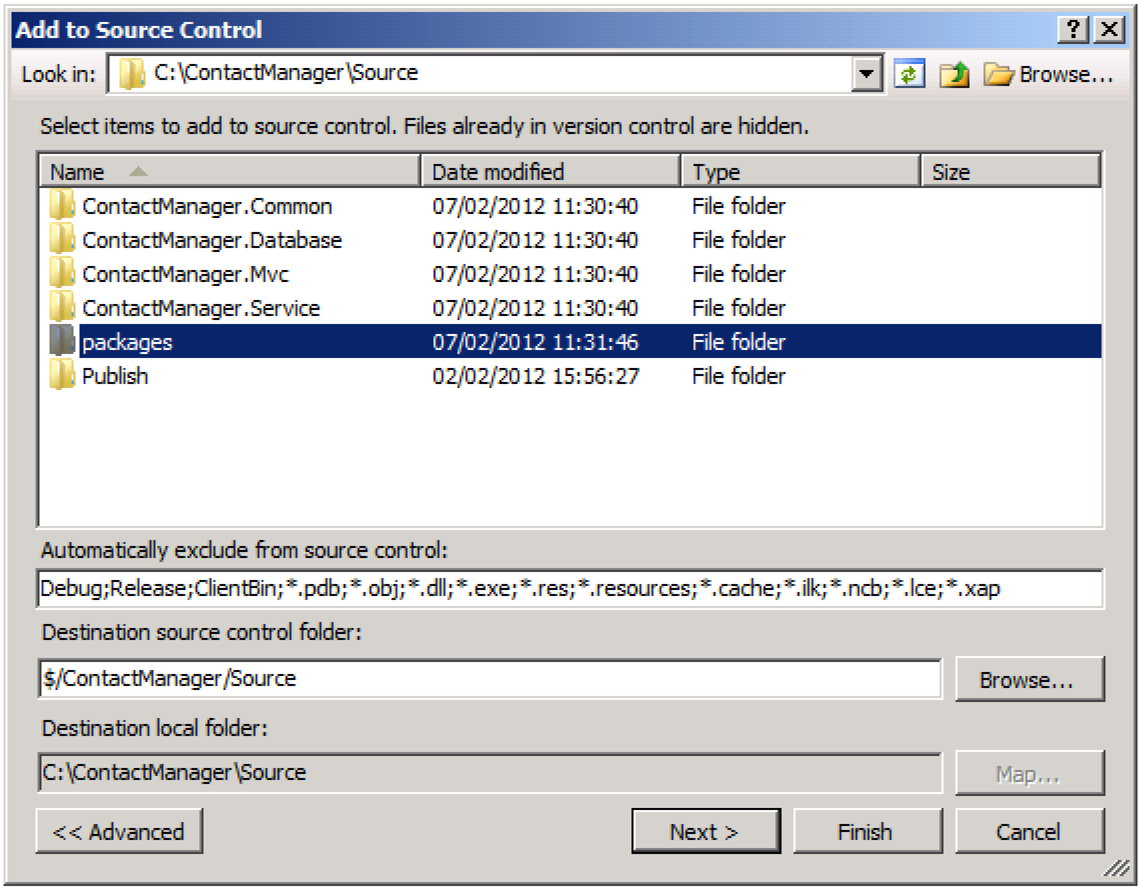


Workspace: DEVWORKSTATION

Source location: $/ContactManager/Source

Local Path: C:\ContactManager\Source

5. In the **Add to Source Control** dialog box, select the folder or items you want to add, and then click **Next**.



Look in: C:\ContactManager\Source

Select items to add to source control. Files already in version control are hidden.

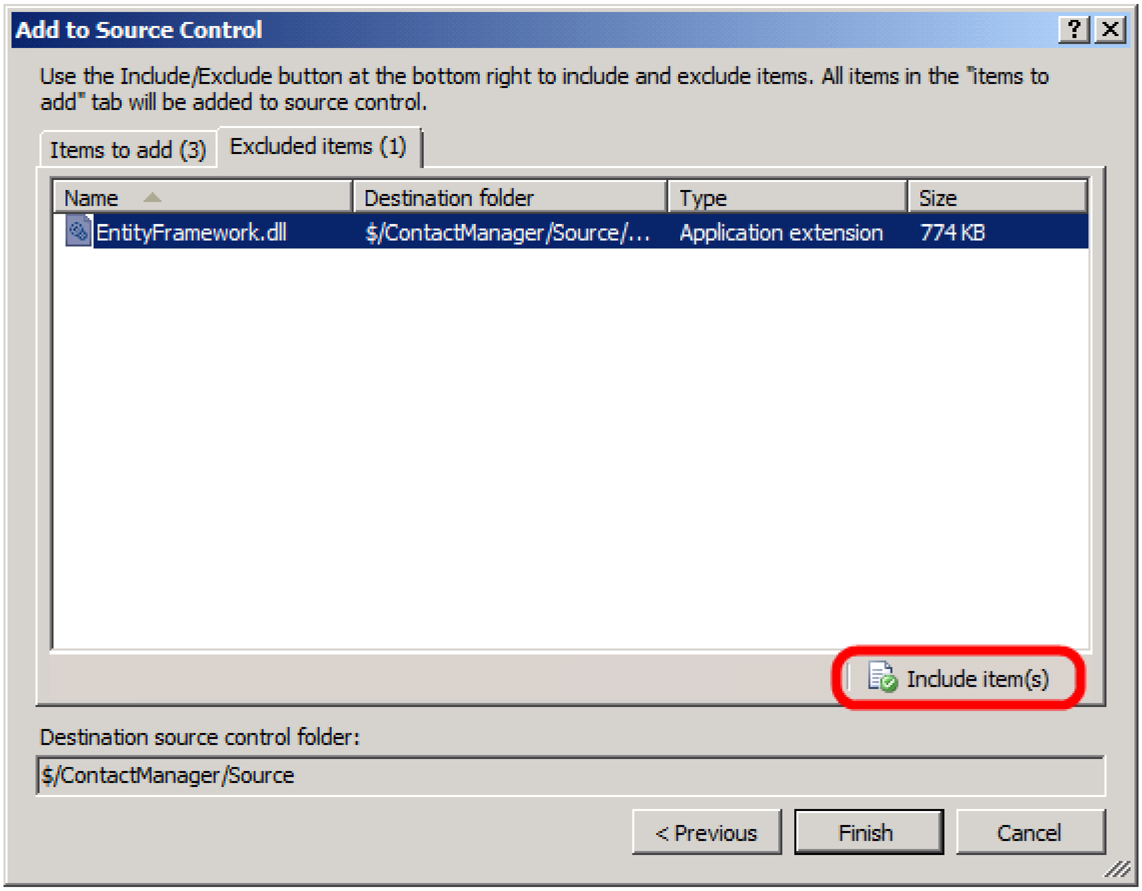
Name	Date modified	Type	Size
ContactManager.Common	07/02/2012 11:30:40	File folder	
ContactManager.Database	07/02/2012 11:30:40	File folder	
ContactManager.Mvc	07/02/2012 11:30:40	File folder	
ContactManager.Service	07/02/2012 11:30:40	File folder	
packages	07/02/2012 11:31:46	File folder	
Publish	02/02/2012 15:56:27	File folder	

Automatically exclude from source control: Debug;Release;ClientBin;*.pdb;*.obj;*.dll;*.exe;*.res;*.resources;*.cache;*.ilk;*.ncb;*.lce;*.xap

Destination source control folder: $/ContactManager/Source

Destination local folder: C:\ContactManager\Source

6. On the **Excluded items** tab, select any required items that have been automatically excluded (for example, assemblies), and then click **Include item(s)**.



Add to Source Control

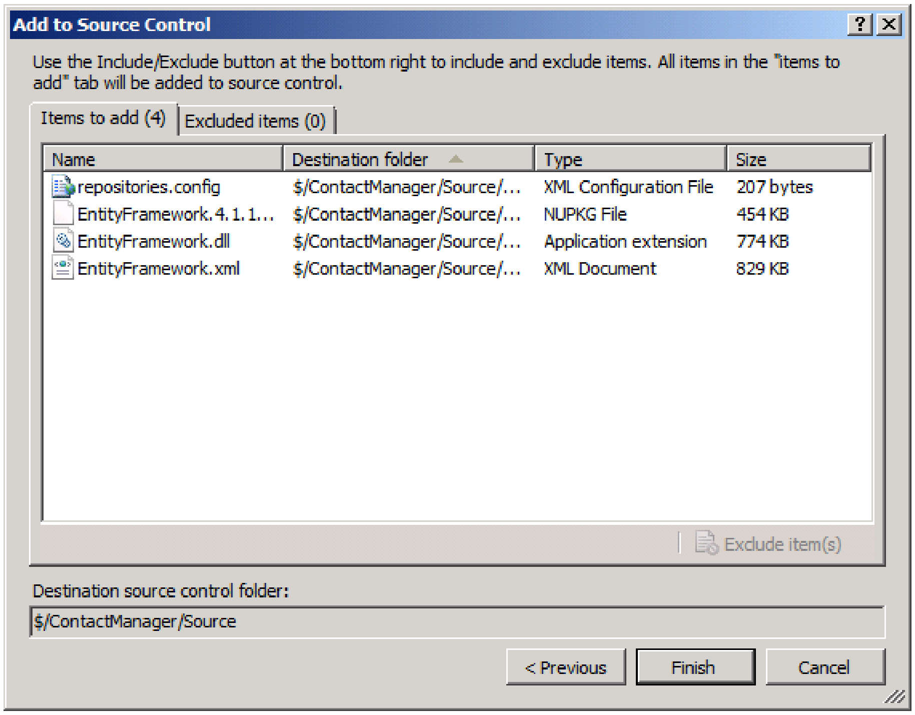
Use the Include/Exclude button at the bottom right to include and exclude items. All items in the "items to add" tab will be added to source control.

Name	Destination folder	Type	Size
EntityFramework.dll	$/ContactManager/Source/...	Application extension	774 KB

Destination source control folder:

$/ContactManager/Source

7. On the **Items to add** tab, verify that all the files you want to include are listed, and then click **Finish**.



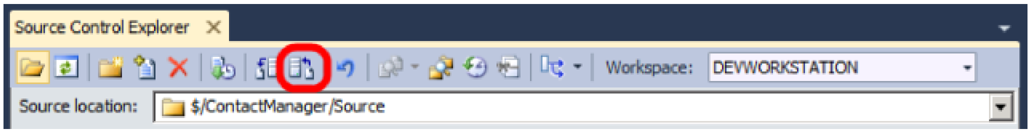
Add to Source Control

Use the Include/Exclude button at the bottom right to include and exclude items. All items in the "items to add" tab will be added to source control.

Name	Destination folder	Type	Size
repositories.config	$/ContactManager/Source/...	XML Configuration File	207 bytes
EntityFramework.4.1.1...	$/ContactManager/Source/...	NUPKG File	454 KB
EntityFramework.dll	$/ContactManager/Source/...	Application extension	774 KB
EntityFramework.xml	$/ContactManager/Source/...	XML Document	829 KB

Destination source control folder: $/ContactManager/Source

8. In the **Source Control Explorer** window, click the **Check In** button.



Workspace: DEVWORKSTATION

Source location: $/ContactManager/Source

9. In the **Check In – Source Files** dialog box, type a comment, and then click **Check In**.

At this point, you have added the external dependencies for your solution to source control.

Conclusion

This topic described how to connect to a team project, map a folder structure, and add content to source control. For more information on how to work with items under source control, see Using Version Control.

The next topic, Configuring a TFS Build Server for Web Deployment, describes how to prepare a TFS Team Build server to build and deploy your solution.

Further Reading

For more comprehensive information on working with source control in TFS, see Using Version Control.

Configuring a TFS Build Server for Web Deployment

This topic describes how to prepare a Team Foundation Server (TFS) build server to build and deploy your solutions using Team Build and the Internet Information Services (IIS) Web Deployment Tool (Web Deploy).

Task Overview

To prepare a build server to build and deploy your solutions, you'll need to:

- Install and configure the TFS build service.
- Install Visual Studio 2010.
- Install any products or components that are required to build your solution, like versions of the .NET Framework or ASP.NET MVC.
- Install Web Deploy 2.0 or later.

This topic will show you how to perform these procedures or point to other resources where they exist. The tasks and walkthroughs in this topic assume that:

- You're starting with a clean server build running Windows Server 2008 R2 Service Pack 1.
- The server is domain-joined with a static IP address.
- You've installed the TFS application tier on a separate server, as described in Enterprise Web Deployment: Scenario Overview.

Who Performs These Procedures?

In most cases, a TFS administrator will be responsible for configuring build servers. In some cases, the developer team may take ownership of specific build servers.

Install and Configure the TFS Build Service

When you configure a build server, your first task is to install and configure the TFS build service. As part of this process, you'll need to:

- Install the TFS build service and configure a service account. Any build tasks, including deployment, will run using the identity of the build service account.
- Create a *build controller* and one or more *build agents*. Each build controller manages a set of build agents. When you queue a build, the build controller assigns the build task to an available build agent. Each team project collection in TFS is mapped to a single build controller.
- Configure a drop folder for your build outputs. This is a network share. Any build outputs, like web deployment packages, are sent to the drop folder.

The Administering Team Foundation Build chapter on MSDN contains all the resources you need in order to perform these tasks:

- For a conceptual overview of Team Foundation Build, including the build service, build controllers, and build agents, see Understanding a Team Foundation Build System.
- For information on installing and configuring the build service, see Configure a Build Machine.
- For information on creating build controllers, see Create and Work with a Build Controller.
- For information on creating build agents, see Create and Work with Build Agents.
- For information on creating and configuring drop folders, see Set Up Drop Folders.

Install Required Products and Components

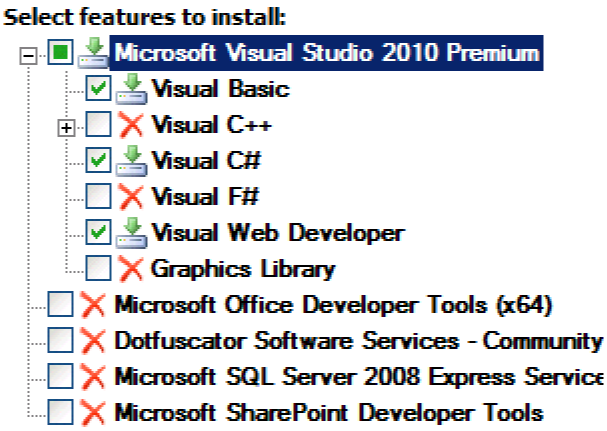
To enable the build server to build your solutions, you must install any products, components, or assemblies that your solution requires. Before you install any web platform components, you should install Visual Studio 2010 (any version) on the build server. This ensures that the core Microsoft Build Engine (MSBuild) target files and the Web Publishing Pipeline (WPP) target files are available to the build service. The Visual Studio installer should also install Web Deploy, which you'll need if you plan to deploy web packages as part of your build process.

The best way to install common web platform components is to use the Web Platform Installer. This ensures that you're installing the latest version of each product, and it also automatically detects and installs any prerequisites for each product. In the case of the Contact Manager solution, you should use the Web Platform Installer to install these products and components:

- **.NET Framework 4.0**. This is required to run applications that were built on this version of the .NET Framework.
- **Web Deployment Tool 2.1 or later**. This installs Web Deploy (and its underlying executable, MSDeploy.exe) on your server. As part of this process, it installs and starts the Web Deployment Agent Service. This service lets you deploy web packages from a remote computer.
- **ASP.NET MVC 3**. This installs the assemblies you need to run ASP.NET MVC 3 applications.

To install the required products and components

1. Install Visual Studio 2010. When prompted to select features to install, you should include:
    a. Any programming languages that you need to compile.
    b. Visual Web Developer. This ensures that the WPP targets are added to your build server.



Select features to install:

- ■ Microsoft Visual Studio 2010 Premium
  - ☑ Visual Basic
  - ☐ Visual C++
  - ☑ Visual C#
  - ☐ Visual F#
  - ☑ Visual Web Developer
  - ☐ Graphics Library
- ☐ Microsoft Office Developer Tools (x64)
- ☐ Dotfuscator Software Services - Community
- ☐ Microsoft SQL Server 2008 Express Service
- ☐ Microsoft SharePoint Developer Tools

2. When the installation of Visual Studio 2010 is complete, download and install Visual Studio 2010 Service Pack 1 (if it's not already included in your installation media).

> **Note:** Visual Studio 2010 Service Pack 1 resolves a bug that can prevent MSBuild from locating the MSDeploy executable.

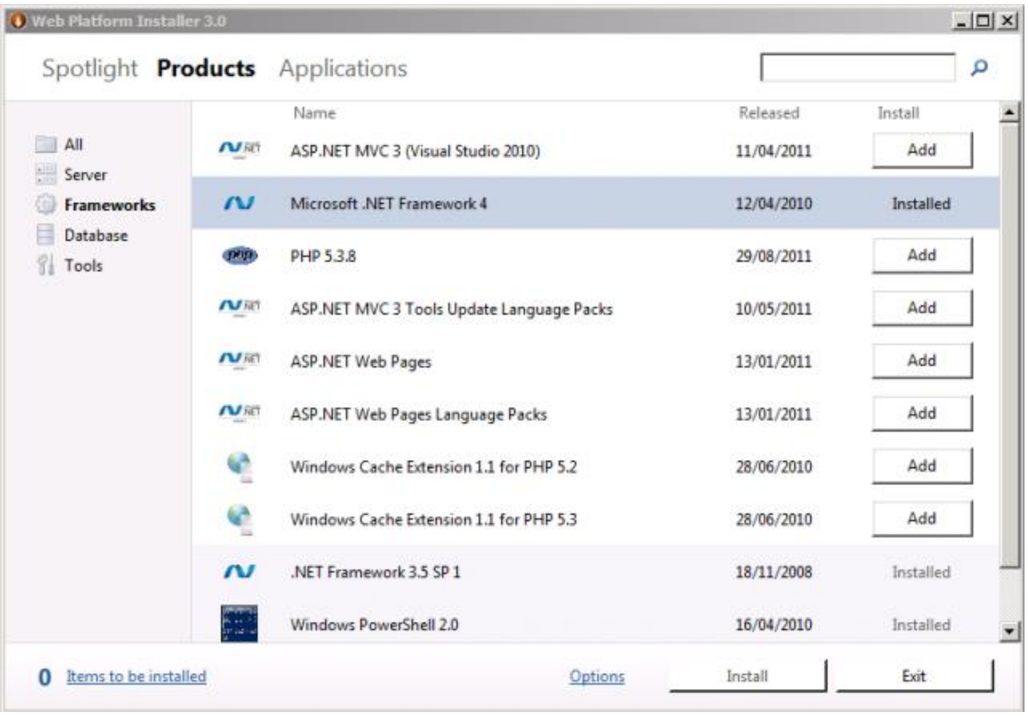
3. Download and launch the Web Platform Installer.

4. At the top of the **Web Platform Installer 3.0** window, click **Products**.

5. On the left side of the window, in the navigation pane, click **Frameworks**.

6. In the **Microsoft .NET Framework 4** row, if the .NET Framework is not already installed, click **Add**.

> **Note:** You may have already installed the .NET Framework 4.0 through Windows Update. If a product or component is already installed, the Web Platform Installer will indicate this by replacing the **Add** button with the text **Installed**.



Name	Released	Install
ASP.NET MVC 3 (Visual Studio 2010)	11/04/2011	Add
Microsoft .NET Framework 4	12/04/2010	Installed
PHP 5.3.8	29/08/2011	Add
ASP.NET MVC 3 Tools Update Language Packs	10/05/2011	Add
ASP.NET Web Pages	13/01/2011	Add
ASP.NET Web Pages Language Packs	13/01/2011	Add
Windows Cache Extension 1.1 for PHP 5.2	28/06/2010	Add
Windows Cache Extension 1.1 for PHP 5.3	28/06/2010	Add
.NET Framework 3.5 SP 1	18/11/2008	Installed
Windows PowerShell 2.0	16/04/2010	Installed

7. In the **ASP.NET MVC 3 (Visual Studio 2010)** row, click **Add**.
8. In the navigation pane, click **Server**.
9. In the **Web Deployment Tool 2.1** row, click **Add**.

   Click **Install**. The Web Platform Installer will show you a list of products—together with any associated dependencies—to be installed and will prompt you to accept the license terms.
10. Review the license terms, and if you consent to the terms, click **I Accept**.
11. When the installation is complete, click **Finish**, and then close the **Web Platform Installer 3.0** window.

Note: If your deployment process includes the use of tools like VSDBCMD.exe or SQLCMD.exe, you'll need to ensure that these are installed on your build server. VSDBCMD.exe is a Visual Studio tool and is typically added to the server when you install Team Foundation Build. SQLCMD.exe is a SQL Server tool. You can download a stand-alone version of SQLCMD.exe from the Microsoft SQL Server 2008 R2 Feature Pack page.

Conclusion

At this point, your build server is ready to start building and deploying your web application projects. The next topic, Creating a Build Definition That Supports Deployment, describes how to create and configure a build definition to control when and how your projects are built and deployed.

Further Reading

For more general guidance on working with Team Build, see Administering Team Foundation Build.

Creating a Build Definition That Supports Deployment

If you want to perform any kind of build in Team Foundation Server (TFS) 2010, you need to create a build definition within your team project. This topic describes how to create a new build definition in TFS and how to control web deployment as part of the build process in Team Build.

Task Overview

A build definition is the mechanism that controls how and when builds occur for team projects in TFS. Each build definition specifies:

- The things you want to build, like Visual Studio solution files or custom Microsoft Build Engine (MSBuild) project files.
- The criteria that determine when a build should take place, like manual triggers, continuous integration (CI), or gated check-ins.
- The location to which Team Build should send build outputs, including deployment artifacts like web packages and database scripts.
- The amount of time that each build should be retained.
- Various other parameters of the build process.

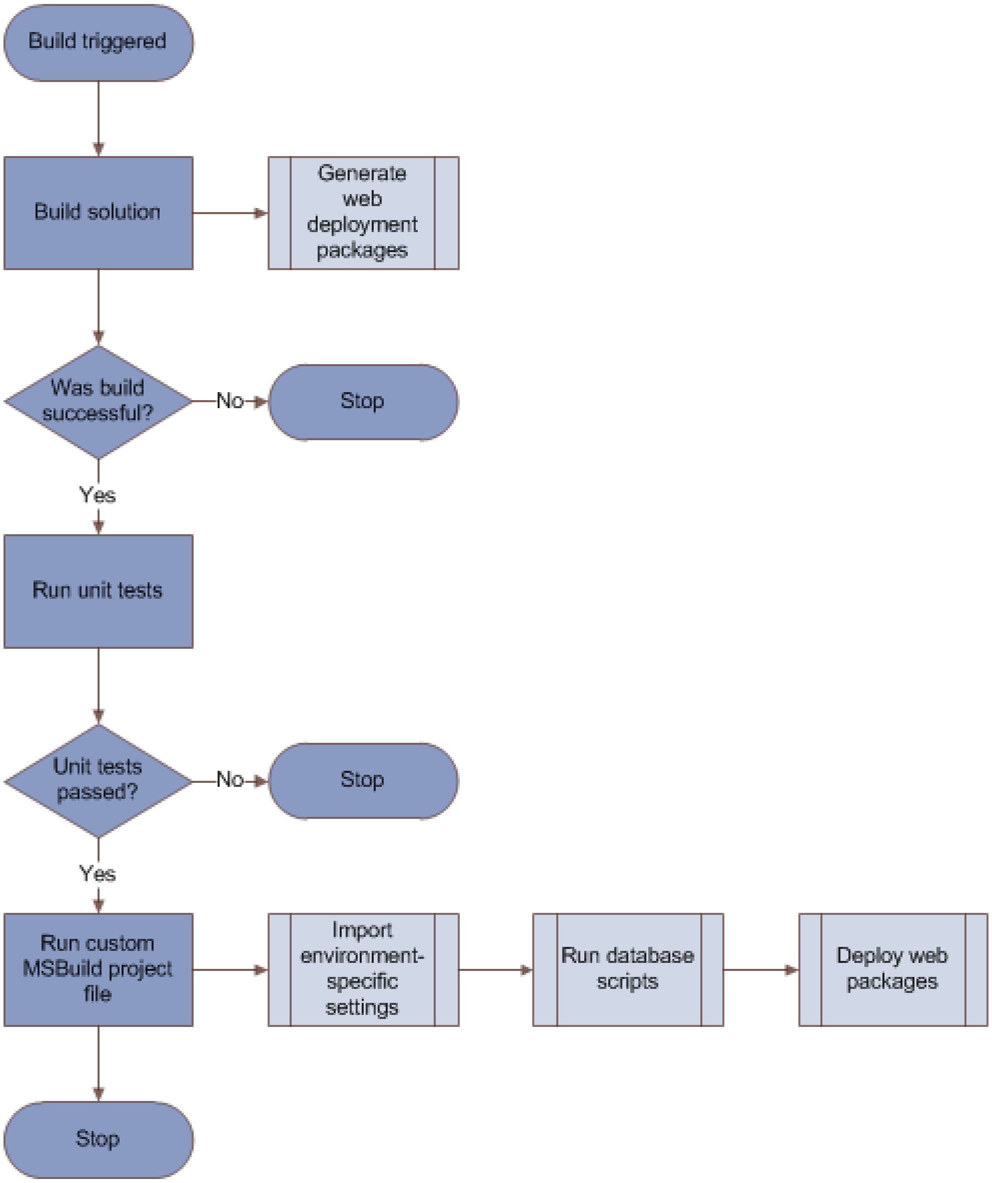
Note: For more information on build definitions, see Define Your Build Process.

This topic will show you how to create a build definition that uses CI, so that a build is triggered when a developer checks in new content. If the build succeeds, the build service runs a custom project file to deploy the solution to a test environment.

When you trigger a build, these actions need to happen:

- First, Team Build should build the solution. As part of this process, Team Build will invoke the Web Publishing Pipeline (WPP) to generate web deployment packages for each of the web application projects in the solution. Team Build will also run any unit tests associated with the solution.
- If the solution build fails, Team Build should take no further action. Unit test failures should be treated as a build failure.
- If the solution build succeeds, Team Build should run the custom project file that controls the deployment of the solution. As part of this process, Team Build will invoke the Internet Information Services (IIS) Web Deployment Tool (Web Deploy) to install the packaged web applications on the destination web servers, and it will invoke the VSDBCMD.exe utility to run database creation scripts on the destination database servers.

This illustrates the process:



The Contact Manager sample solution includes a custom MSBuild project file, *Publish.proj*, that you can run from MSBuild or Team Build. As described in Understanding the Build Process, this project file defines the logic that deploys your web packages and databases to a target environment. The file includes logic that omits the building and packaging process if it's running in Team Build, leaving just the deployment tasks to run. This is because when you automate deployment in this way, you'll typically want to ensure that the solution builds successfully and passes any unit tests before the deployment process commences.

The next section explains how to implement this process by creating a new build definition.

Note: This procedure—in which a single automated process builds, tests, and deploys a solution—is likely to be most suited to deployment to test environments. For staging and production environments you're a lot more likely to want to deploy content from a previous build that you've already verified and validated in a test environment. This approach is described in the next topic, Deploying a Specific Build.

Who Performs This Procedure?

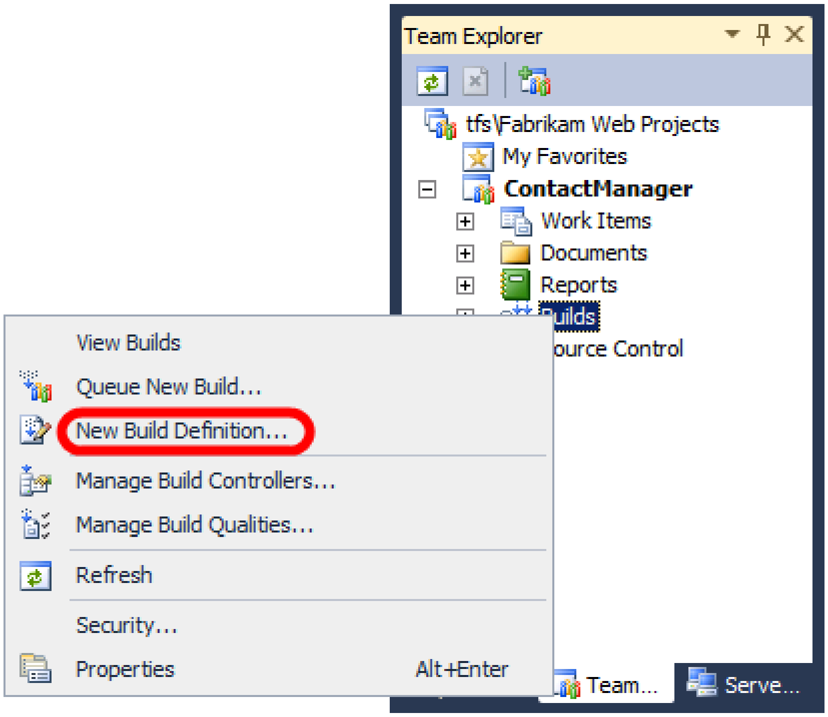
Typically, a TFS administrator performs this procedure. In some cases, a developer team leader may take responsibility for the team project collection in TFS. In order to create a new build definition, you need to be a member of the **Project Collection Build Administrators** group for the team project collection that contains your solution.

Create a Build Definition for CI and Deployment

The next procedure describes how to create a build definition that CI triggers. If the build succeeds, the solution is deployed using the logic in a custom MSBuild project file.

To create a build definition for CI and deployment

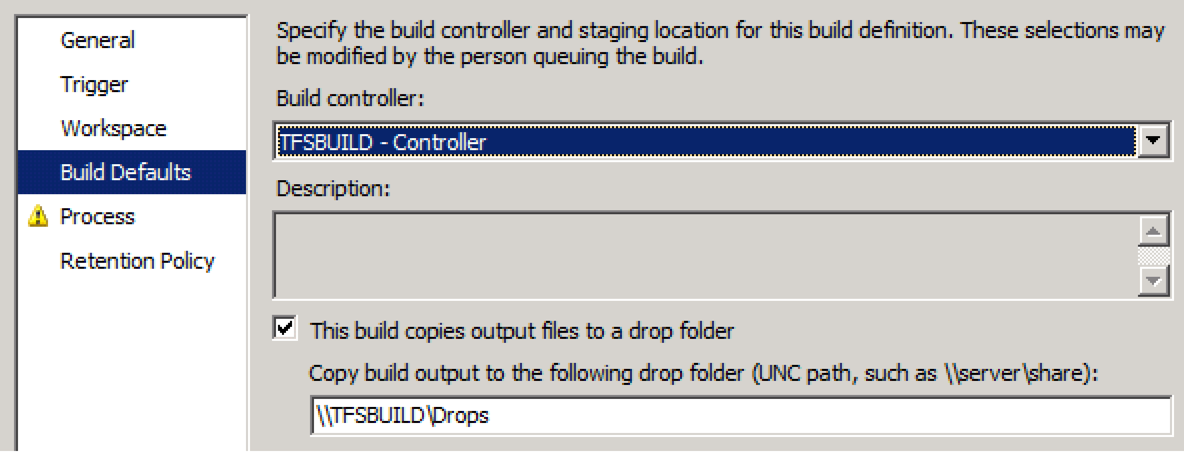
1. In Visual Studio 2010, in the **Team Explorer** window, expand your team project node, right-click **Builds**, and then click **New Build Definition**.



2. On the **General** tab, give the build definition a name (for example, **DeployToTest**) and an optional description.

3. On the **Trigger** tab, select the criteria on which you want to trigger a new build. For example, if you want to build the solution and deploy to the test environment every time a developer checks in new code, select **Continuous Integration**.

4. On the **Build Defaults** tab, in the **Copy build output to the following drop folder** box, type the Universal Naming Convention (UNC) path of your drop folder (for example, **\\TFSBUILD\Drops**).



Specify the build controller and staging location for this build definition. These selections may be modified by the person queuing the build.

Build controller: TFSBUILD - Controller

Description:

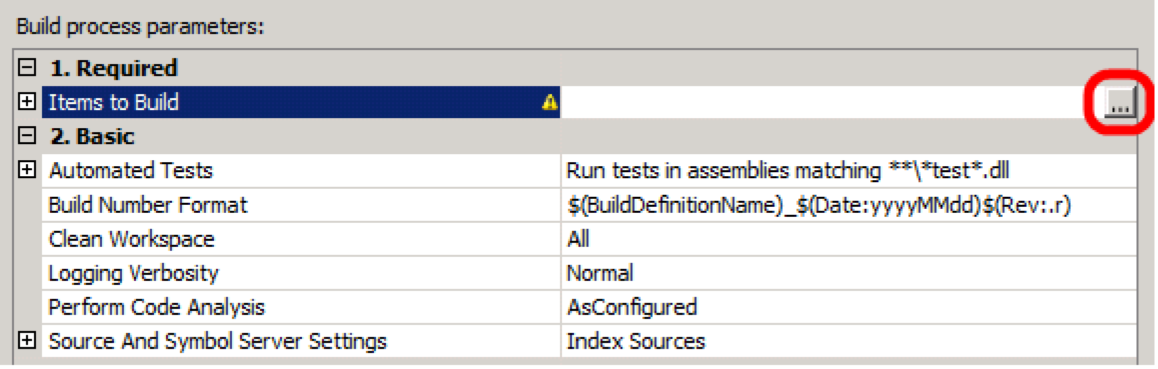
☑ This build copies output files to a drop folder

Copy build output to the following drop folder (UNC path, such as \\server\share): \\TFSBUILD\Drops

> **Note:** This drop location stores several builds, depending on the retention policy you configure. When you want to publish deployment artifacts from a specific build to a staging or production environment, this is where you'll find them.

5. On the **Process** tab, in the **Build process file** dropdown list, leave **DefaultTemplate.xaml** selected. This is one of the default build process templates that get added to all new team projects.

6. In the **Build process parameters** table, click in the **Items to Build** row, and then click the **ellipsis** button.

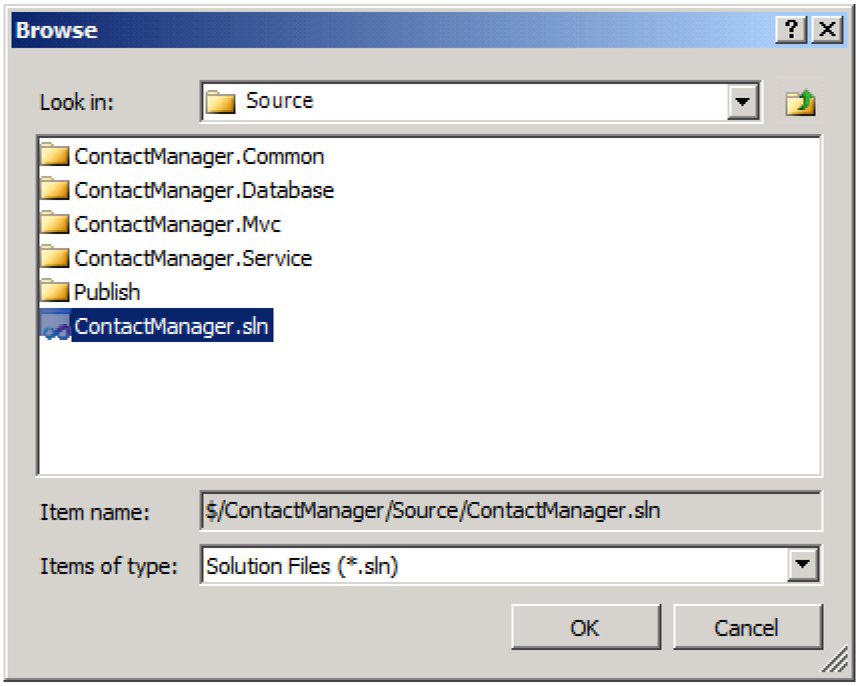


Build process parameters:

⊟ **1. Required**	
⊞ Items to Build ⚠	...
⊟ **2. Basic**	
⊞ Automated Tests	Run tests in assemblies matching ***test*.dll
Build Number Format	$(BuildDefinitionName)_$(Date:yyyyMMdd)$(Rev:.r)
Clean Workspace	All
Logging Verbosity	Normal
Perform Code Analysis	AsConfigured
⊞ Source And Symbol Server Settings	Index Sources

7. In the **Items to Build** dialog box, click **Add**.

8. Browse to the location of your solution file, and then click **OK**.


Browse

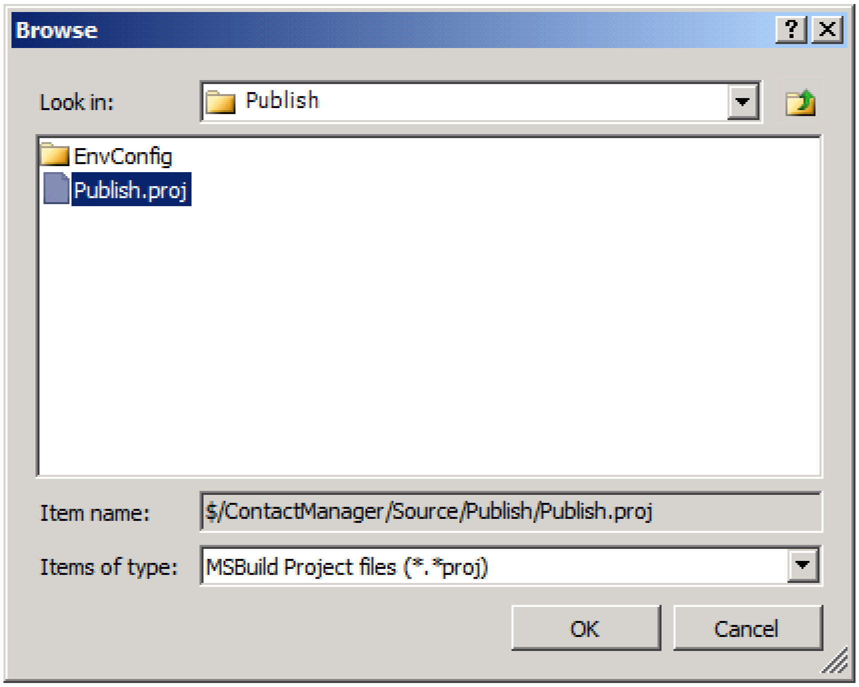
Look in: Source

ContactManager.Common
ContactManager.Database
ContactManager.Mvc
ContactManager.Service
Publish
ContactManager.sln

Item name: $/ContactManager/Source/ContactManager.sln

Items of type: Solution Files (*.sln)

9. In the **Items to Build** dialog box, click **Add**.
10. In the **Items of type** dropdown list, select **MSBuild Project files**.
11. Browse to the location of the custom project file with which you control the deployment process, select the file, and then click **OK**.


Browse

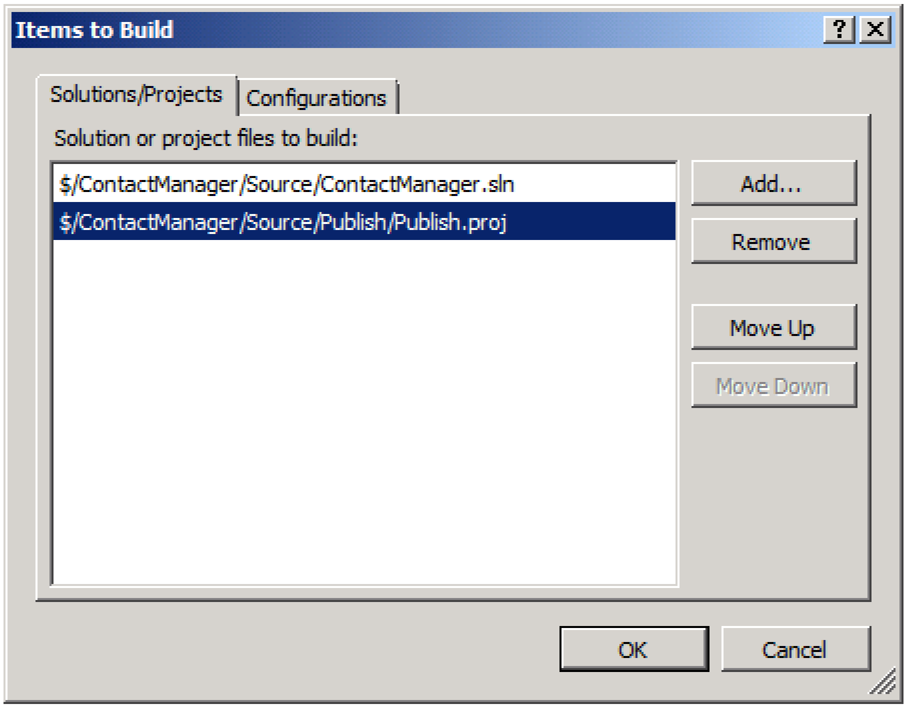
Look in: Publish

EnvConfig
Publish.proj

Item name: $/ContactManager/Source/Publish/Publish.proj

Items of type: MSBuild Project files (*.*proj)

12. The **Items to Build** dialog box should now show two items. Click **OK**.



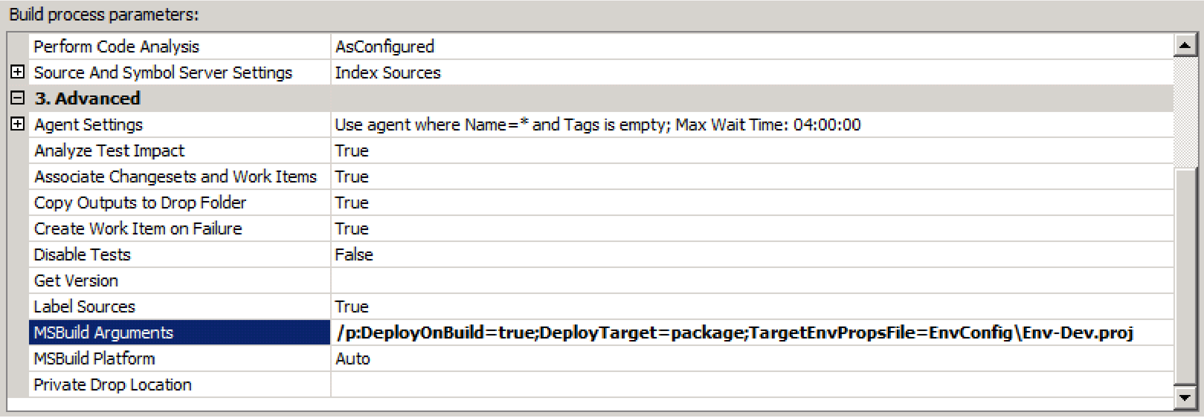
Solution or project files to build:

$/ContactManager/Source/ContactManager.sln
$/ContactManager/Source/Publish/Publish.proj

13. On the **Process** tab, in the **Build process parameters** table, expand the **Advanced** section.

14. In the **MSBuild Arguments** row, add any MSBuild command-line arguments that *either* of your items to build requires. In the Contact Manager solution scenario, these arguments are required:

```
/p:DeployOnBuild=true;DeployTarget=Package;
   TargetEnvPropsFile=EnvConfig\Env-Dev.proj
```



Build process parameters:

Perform Code Analysis	AsConfigured
⊞ Source And Symbol Server Settings	Index Sources
⊟ **3. Advanced**	
⊞ Agent Settings	Use agent where Name=* and Tags is empty; Max Wait Time: 04:00:00
Analyze Test Impact	True
Associate Changesets and Work Items	True
Copy Outputs to Drop Folder	True
Create Work Item on Failure	True
Disable Tests	False
Get Version	
Label Sources	True
MSBuild Arguments	**/p:DeployOnBuild=true;DeployTarget=package;TargetEnvPropsFile=EnvConfig\Env-Dev.proj**
MSBuild Platform	Auto
Private Drop Location	

In this example:

a. The **DeployOnBuild=true** and **DeployTarget=package** arguments are required when you build the Contact Manager solution. This instructs MSBuild to create web deployment packages after building each web application project, as described in Building and Packaging Web Application Projects.

b. The **TargetEnvPropsFile** argument is required when you build the *Publish.proj* file. This property indicates the location of the environment-specific configuration file, as described in Understanding the Build Process.

15. On the **Retention Policy** tab, configure how many builds of each type you want to retain as required.
16. Click **Save**.

Queue a Build

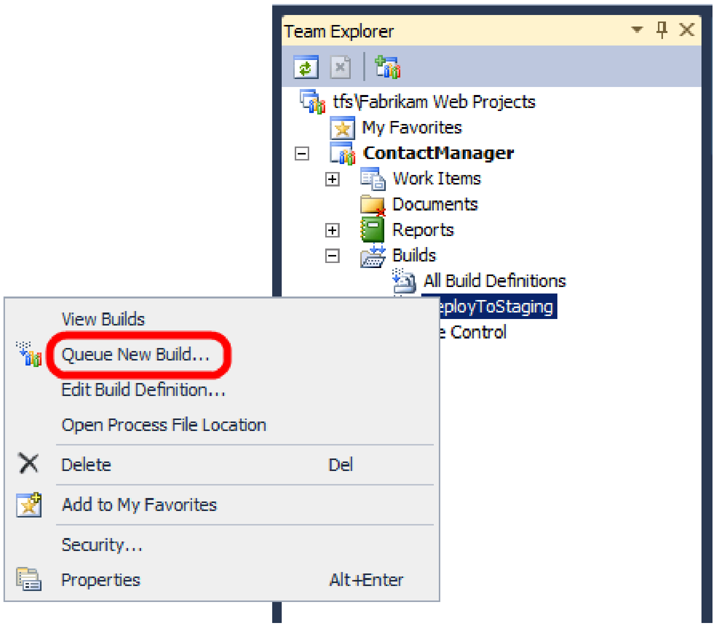
At this point, you have created at least one new build definition. The build process you defined will now run according to the triggers you specified in the build definition.

If you've configured your build definition to use CI, you can test your build definition in two ways:

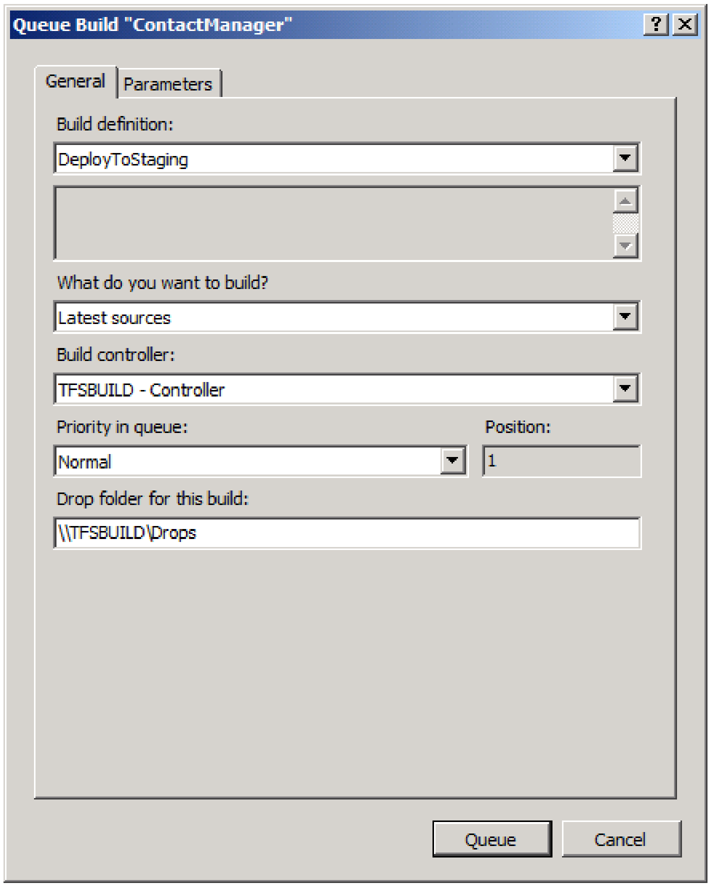
- Check in some content to the team project to trigger an automatic build.
- Queue a build manually.

To queue a build manually

1. In the **Team Explorer** window, right-click the build definition, and then click **Queue New Build**.



2. In the **Queue Build** dialog box, review the build properties, and then click **Queue**.



Build definition: DeployToStaging

What do you want to build? Latest sources

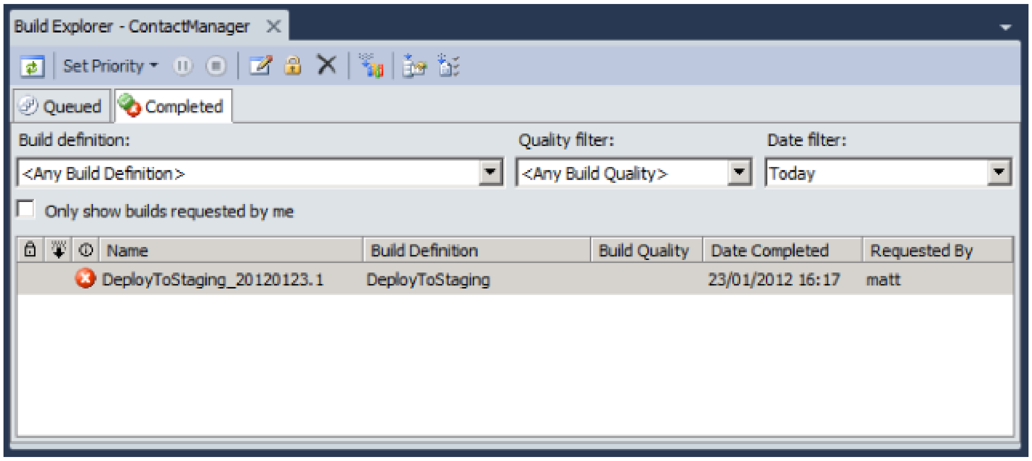
Build controller: TFSBUILD - Controller

Priority in queue: Normal

Position: 1

Drop folder for this build: \\TFSBUILD\Drops

To review the progress and the outcome of a build—regardless of whether it was triggered manually or automatically—double-click the build definition in the **Team Explorer** window. This will open a **Build Explorer** tab.



Build definition: <Any Build Definition>

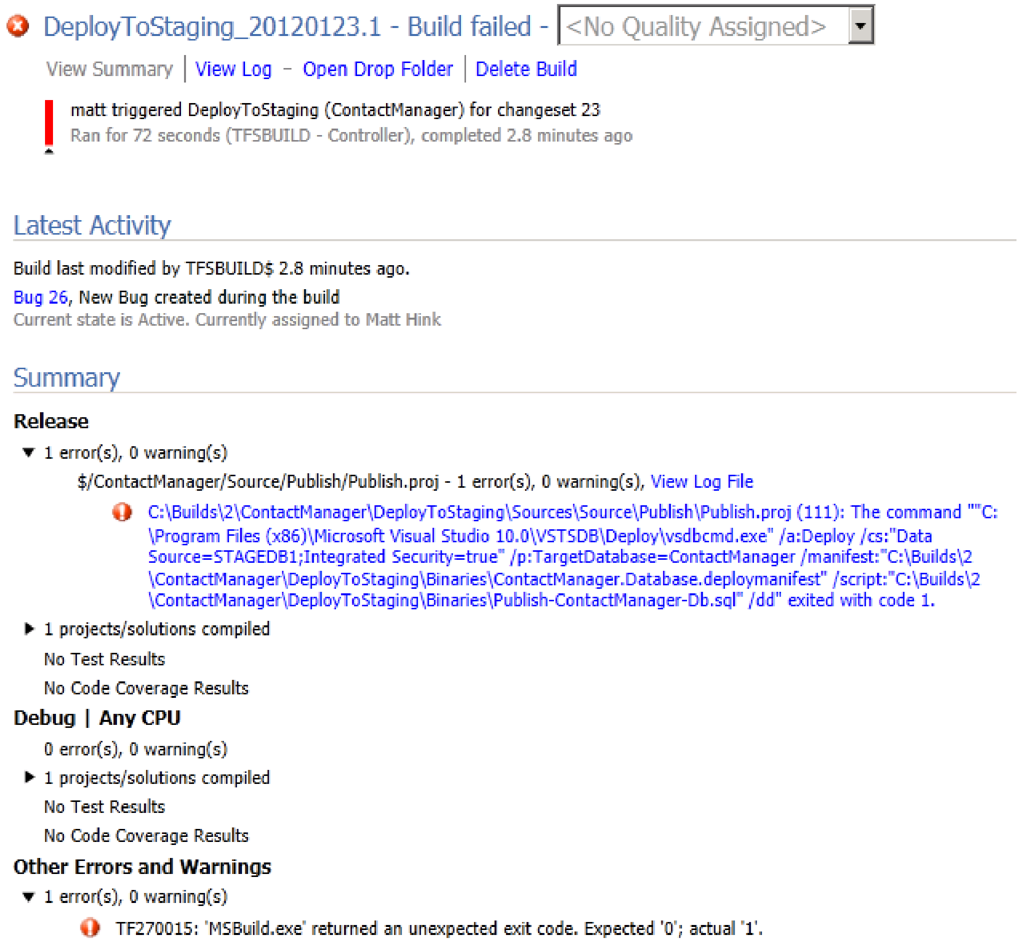
Quality filter: <Any Build Quality>

Date filter: Today

☐ Only show builds requested by me

Name	Build Definition	Build Quality	Date Completed	Requested By
DeployToStaging_20120123.1	DeployToStaging		23/01/2012 16:17	matt

From here, you can troubleshoot failed builds. If you double-click an individual build, you can view summary information and click through to detailed log files.



DeployToStaging_20120123.1 - Build failed - <No Quality Assigned>
View Summary | View Log - Open Drop Folder | Delete Build

matt triggered DeployToStaging (ContactManager) for changeset 23
Ran for 72 seconds (TFSBUILD - Controller), completed 2.8 minutes ago

Latest Activity

Build last modified by TFSBUILD$ 2.8 minutes ago.
Bug 26, New Bug created during the build
Current state is Active. Currently assigned to Matt Hink

Summary

Release

▼ 1 error(s), 0 warning(s)
$/ContactManager/Source/Publish/Publish.proj - 1 error(s), 0 warning(s), View Log File

```
C:\Builds\2\ContactManager\DeployToStaging\Sources\Source\Publish\Publish.proj (111): The command ""C:
\Program Files (x86)\Microsoft Visual Studio 10.0\VSTSDB\Deploy\vsdbcmd.exe" /a:Deploy /cs:"Data
Source=STAGEDB1;Integrated Security=true" /p:TargetDatabase=ContactManager /manifest:"C:\Builds\2
\ContactManager\DeployToStaging\Binaries\ContactManager.Database.deploymanifest" /script:"C:\Builds\2
\ContactManager\DeployToStaging\Binaries\Publish-ContactManager-Db.sql" /dd" exited with code 1.
```

▶ 1 projects/solutions compiled
No Test Results
No Code Coverage Results

Debug | Any CPU

0 error(s), 0 warning(s)
▶ 1 projects/solutions compiled
No Test Results
No Code Coverage Results

Other Errors and Warnings

▼ 1 error(s), 0 warning(s)

```
TF270015: 'MSBuild.exe' returned an unexpected exit code. Expected '0'; actual '1'.
```

You can use this information to troubleshoot failed builds and address any problems before you attempt another build.

Note: Builds that execute deployment logic are likely to fail until you have granted the build server any permissions required in the destination environment. For more information, see Configuring Permissions for Team Build Deployment.

Monitor the Build Process

TFS provides a broad range of functionality to help you monitor the build process. For example, TFS can send you an email or display alerts in your taskbar notification area when a build has completed. For more information, see Run and Monitor Builds.

Conclusion

This topic described how to create a build definition in TFS. The build definition is configured for CI, so the build process runs whenever a developer checks in content to the team project. The build definition executes a custom MSBuild project file to deploy web packages and database scripts to a target server environment.

In order for an automated deployment to succeed as part of a build process, you'll need to grant appropriate permissions to the build service account on the target web servers and the target database server. The final topic in this tutorial, Configuring Permissions for Team Build Deployment, describes how to identify and configure the permissions required for automated deployment from a Team Build server.

Further Reading

For more information on creating build definitions, see Create a Basic Build Definition and Define Your Build Process. For more guidance on queuing builds, see Queue a Build.

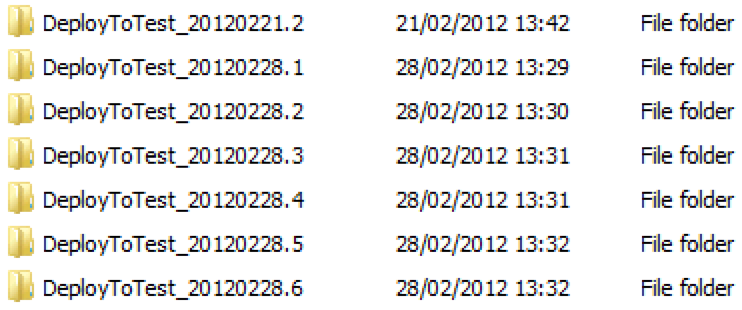
Deploying a Specific Build

This topic describes how to deploy web packages and database scripts from a specific previous build to a new destination, like a staging or production environment.

Task Overview

Until now, the topics in this tutorial set have focused on how to build, package, and deploy web applications and databases as part of a single-step or automated process. However, in some common scenarios, you'll want to select the resources that you deploy from a list of builds in a drop folder. In other words, the latest build may not be the build you want to deploy.

Consider the continuous integration (CI) scenario described in the previous topic, Creating a Build Definition That Supports Deployment. You've created a build definition in Team Foundation Server (TFS) 2010. Every time a developer checks code into TFS, Team Build will build your code, create web packages and database scripts as part of the build process, run any unit tests, and deploy your resources to a test environment. Depending on the retention policy you configured when you created the build definition, TFS will retain a certain number of previous builds.



DeployToTest_20120221.2	21/02/2012 13:42	File folder
DeployToTest_20120228.1	28/02/2012 13:29	File folder
DeployToTest_20120228.2	28/02/2012 13:30	File folder
DeployToTest_20120228.3	28/02/2012 13:31	File folder
DeployToTest_20120228.4	28/02/2012 13:31	File folder
DeployToTest_20120228.5	28/02/2012 13:32	File folder
DeployToTest_20120228.6	28/02/2012 13:32	File folder

Now, suppose you've performed verification and validation testing against one of these builds in your test environment, and you're ready to deploy your application to a staging environment. In the meantime, developers may have checked in new code. You don't want to rebuild the solution and deploy to the staging environment, and you don't want to deploy the latest build to the staging environment. Instead, you want to deploy the specific build that you've verified and validated on the test servers.

To accomplish this, you need to tell the Microsoft Build Engine (MSBuild) where to find the web packages and database scripts that a specific build generated.

Overriding the OutputRoot Property

In the sample solution, the *Publish.proj* file declares a property named **OutputRoot**. As the name suggests, this is the root folder that contains everything that the build process generates. In the *Publish.proj* file, you can see that the **OutputRoot** property refers to the root location for all deployment resources.

> **Note: OutputRoot** is a commonly used property name. Visual C# and Visual Basic project files also declare this property to store the root location for all build outputs.

XML

```xml
<PropertyGroup>
  <!--This is where the .deploymanifest file will be written to during a build-->
  <_DbDeployManifestPath>
    $(OutputRoot)ContactManager.Database.deploymanifest
  </_DbDeployManifestPath>

  <!-- The folder where the .zip and .cmd file will be located for 
       ContactManager.Mvc Web project -->
  <_ContactManagerDest>
    $(OutputRoot)_PublishedWebsites\ContactManager.Mvc_Package\
  </_ContactManagerDest>

  <!-- The folder where the .zip and .cmd file will be located for 
       ContactManager.Service Web project -->
   <_ContactManagerSvcDest>
    $(OutputRoot)_PublishedWebsites\ContactManager.Service_Package\
  </_ContactManagerSvcDest>

  <!-- ... -->
</PropertyGroup>
```

If you want your project file to deploy web packages and database scripts from a different location—like the outputs of a previous TFS build—you simply need to override the **OutputRoot** property. You should set the property value to the relevant build folder on the Team Build server. If you were running MSBuild from the command line, you could specify a value for **OutputRoot** as a command-line argument:

```
msbuild.exe Publish.proj /p:TargetEnvPropsFile=EnvConfig\Env-Dev.proj
  /p:OutputRoot=\\TFSBUILD\Drops\DeployToTest\DeployToTest_20120228.3\
```

In practice, however, you'd also want to skip the **Build** target—there's no point in building your solution if you don't plan to use the build outputs. You could do this by specifying the targets you want to execute from the command line:

```
msbuild.exe Publish.proj /p:TargetEnvPropsFile=EnvConfig\Env-Dev.proj
```

```
/p:OutputRoot=\\TFSBUILD\Drops\DeployToTest\DeployToTest_20120228.3\
/target:GatherPackagesForPublishing;PublishDBPackages;PublishWebPackages
```

However, in most cases, you'll want to build your deployment logic into a TFS build definition. This enables users with the **Queue builds** permission to trigger the deployment from any Visual Studio installation with a connection to the TFS server.

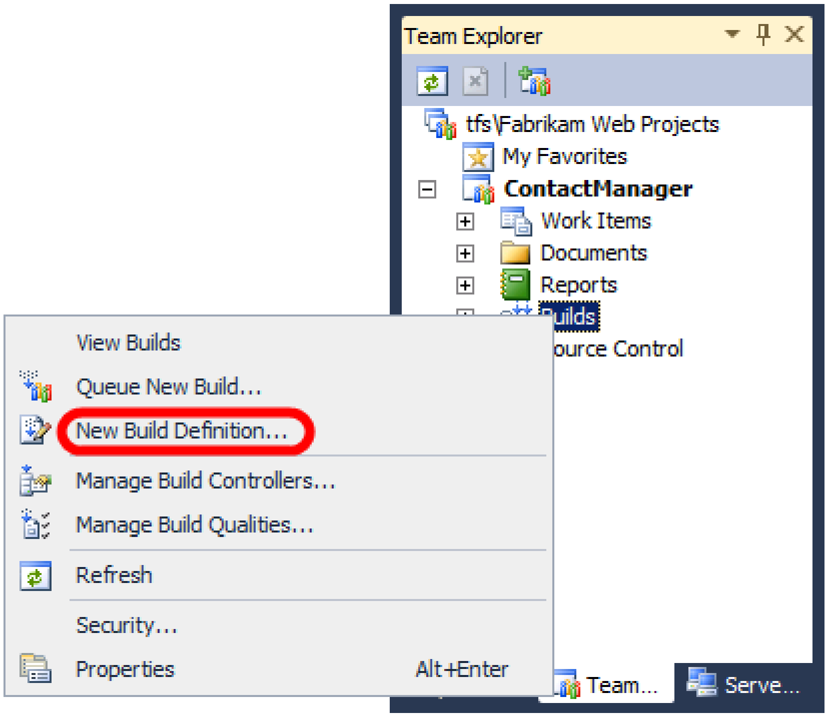
Creating a Build Definition to Deploy Specific Builds

The next procedure describes how to create a build definition that enables users to trigger deployments to a staging environment with a single command.

In this case, you don't want the build definition to actually build anything—you just want it to execute the deployment logic in your custom project file. The *Publish.proj* file includes conditional logic that skips the **Build** target if the file is running in Team Build. It does this by evaluating the built-in **BuildingInTeamBuild** property, which is automatically set to **true** if you run your project file in Team Build. As a result, you can skip the build process and simply run the project file to deploy an existing build.

To create a build definition to trigger deployment manually

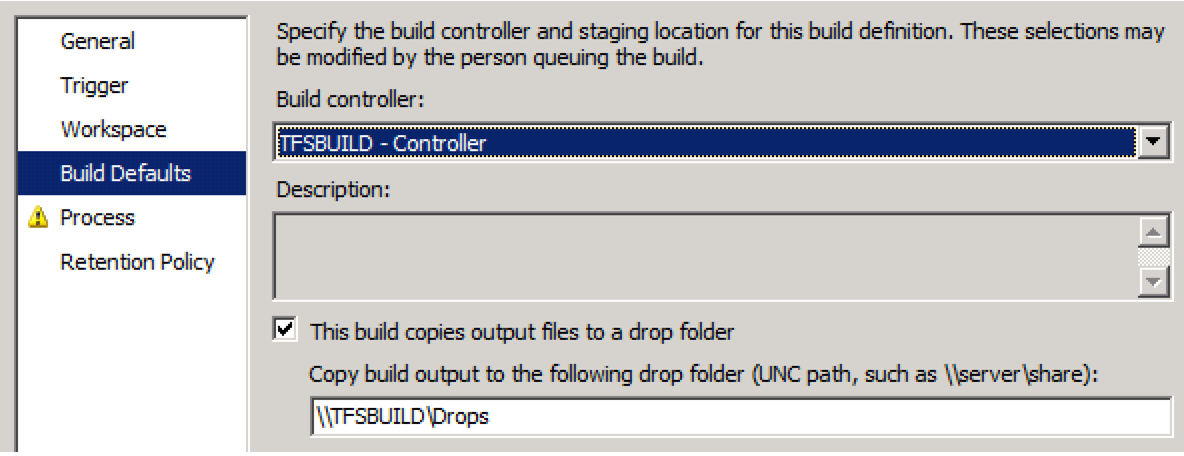
1. In Visual Studio 2010, in the **Team Explorer** window, expand your team project node, right-click **Builds**, and then click **New Build Definition**.



2. On the **General** tab, give the build definition a name (for example, **DeployToStaging**) and an optional description.

3. On the **Trigger** tab, select **Manual – Check-ins do not trigger a new build**.

4. On the **Build Defaults** tab, in the **Copy build output to the following drop folder** box, type the Universal Naming Convention (UNC) path of your drop folder (for example, **\\TFSBUILD\Drops**).



Specify the build controller and staging location for this build definition. These selections may be modified by the person queuing the build.

Build controller: TFSBUILD - Controller

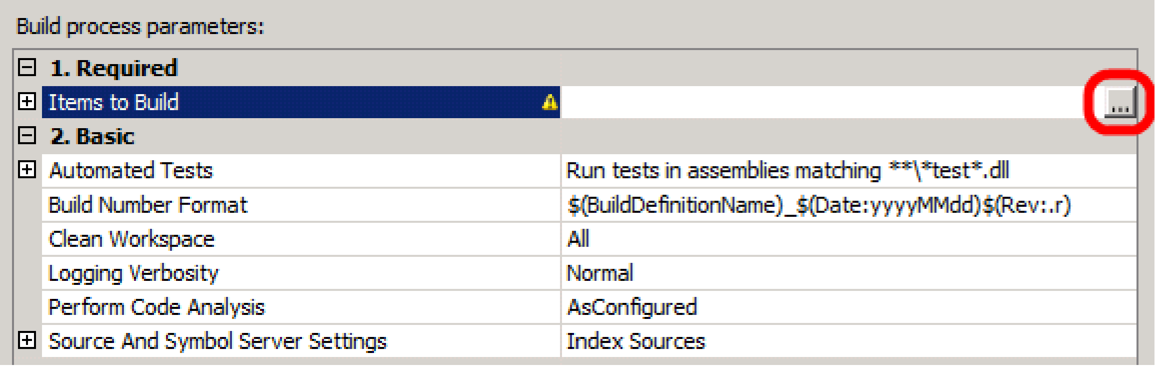
Description:

☑ This build copies output files to a drop folder

Copy build output to the following drop folder (UNC path, such as \\server\share): \\TFSBUILD\Drops

5. On the **Process** tab, in the **Build process file** dropdown list, leave **DefaultTemplate.xaml** selected. This is one of the default build process templates that get added to all new team projects.

6. In the **Build process parameters** table, click in the **Items to Build** row, and then click the **ellipsis** button.



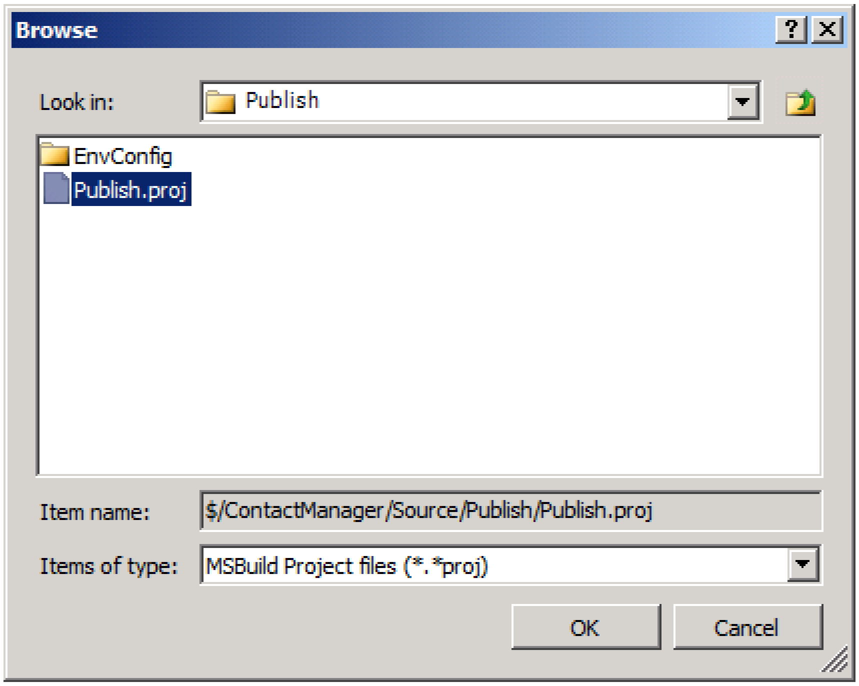
Build process parameters:

⊟ **1. Required**	
⊞ Items to Build	
⊟ **2. Basic**	
⊞ Automated Tests	Run tests in assemblies matching ***test*.dll
Build Number Format	$(BuildDefinitionName)_$(Date:yyyyMMdd)$(Rev:.r)
Clean Workspace	All
Logging Verbosity	Normal
Perform Code Analysis	AsConfigured
⊞ Source And Symbol Server Settings	Index Sources

7. In the **Items to Build** dialog box, click **Add**.

8. In the **Items of type** dropdown list, select **MSBuild Project files**.

9. Browse to the location of the custom project file with which you control the deployment process, select the file, and then click **OK**.



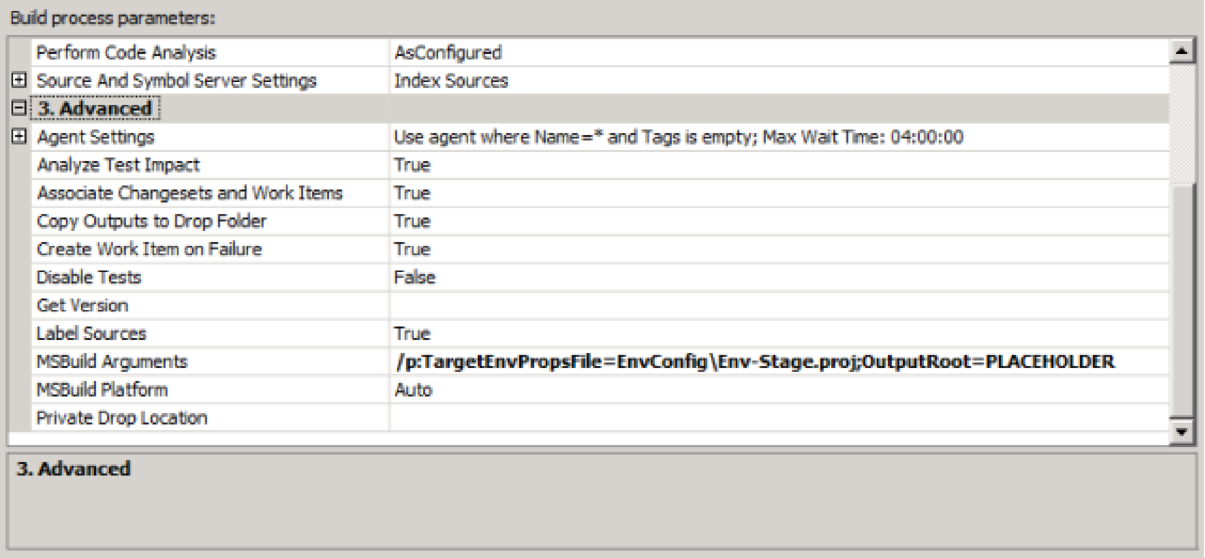
Look in: Publish

Item name: $/ContactManager/Source/Publish/Publish.proj

Items of type: MSBuild Project files (*.*proj)

10. In the **Items to Build** dialog box, click **OK**.
11. In the **Build process parameters** table, expand the **Advanced** section.
12. In the **MSBuild Arguments** row, specify the location of your environment-specific project file and add a placeholder for the location of your build folder:

```
/p:TargetEnvPropsFile=EnvConfig\Env-Stage.proj;
OutputRoot=PLACEHOLDER
```



Build process parameters:

Perform Code Analysis	AsConfigured
⊞ Source And Symbol Server Settings	Index Sources
⊟ **3. Advanced**	
⊞ Agent Settings	Use agent where Name=* and Tags is empty; Max Wait Time: 04:00:00
Analyze Test Impact	True
Associate Changesets and Work Items	True
Copy Outputs to Drop Folder	True
Create Work Item on Failure	True
Disable Tests	False
Get Version	
Label Sources	True
MSBuild Arguments	**/p:TargetEnvPropsFile=EnvConfig\Env-Stage.proj;OutputRoot=PLACEHOLDER**
MSBuild Platform	Auto
Private Drop Location	

3. Advanced

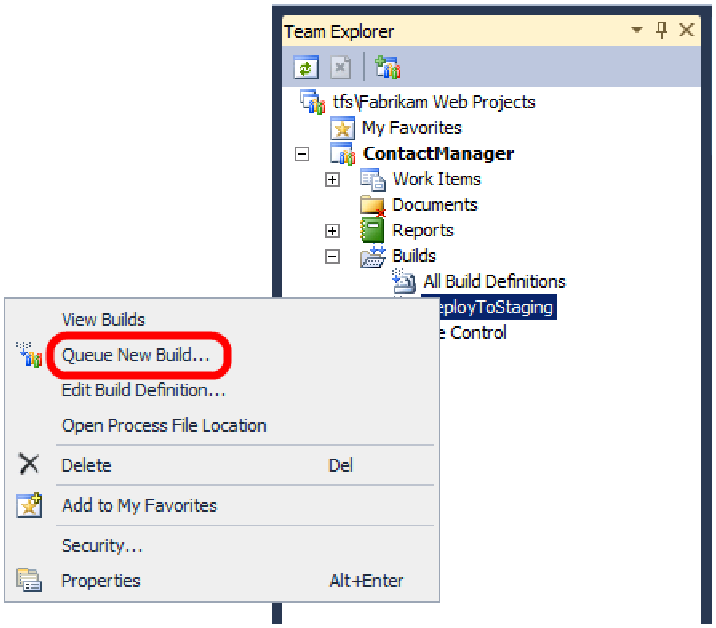
Note: You'll need to override the **OutputRoot** value every time you queue a build. This is covered in the next procedure.

13. Click **Save**.

When you trigger a build, you need to update the **OutputRoot** property to point to the build you want to deploy.

To deploy a specific build from a build definition

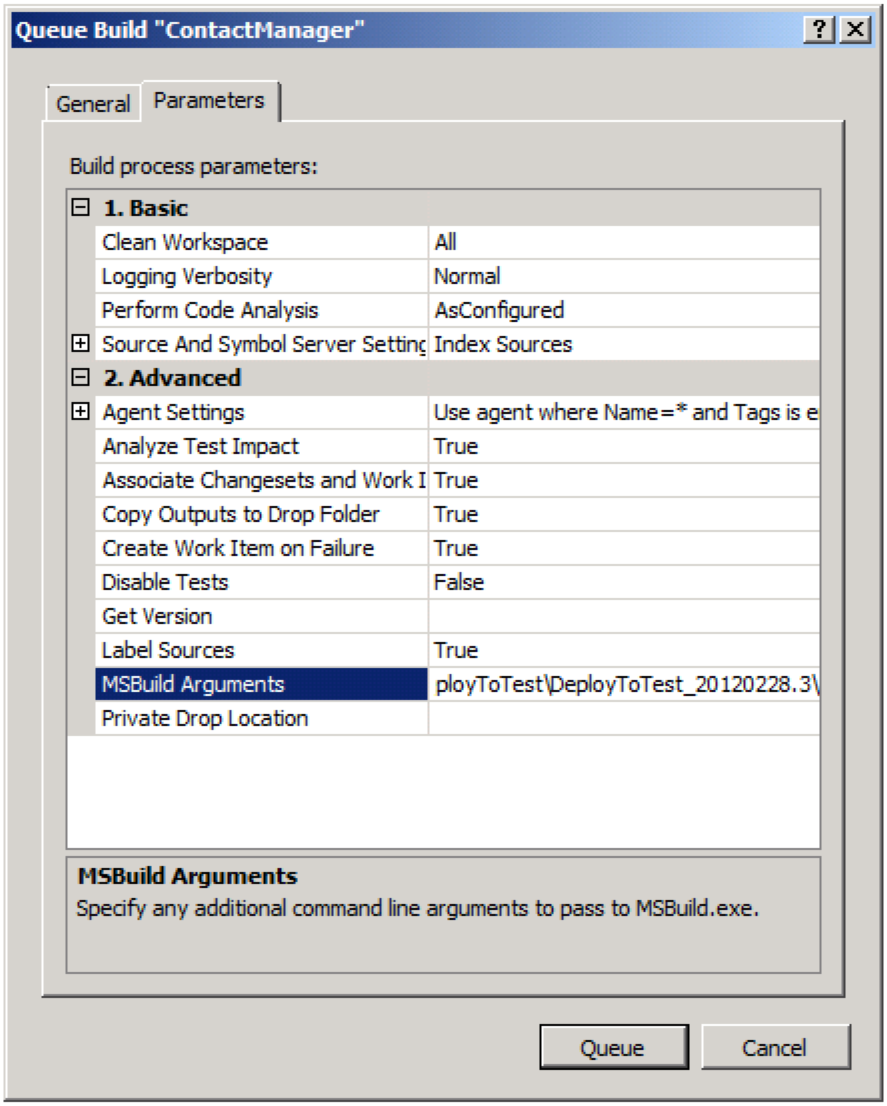
1. In the **Team Explorer** window, right-click the build definition, and then click **Queue New Build**.



2. In the **Queue Build** dialog box, on the **Parameters** tab, expand the **Advanced** section.

3. In the **MSBuild Arguments** row, replace the value of the **OutputRoot** property with the location of your build folder. For example:

```
/p:TargetEnvPropsFile=EnvConfig\Env-Stage.proj;
   OutputRoot=\\TFSBUILD\Drops\DeployToTest\DeployToTest_20120228.3\
```



Build process parameters:

⊟ **1. Basic**	
Clean Workspace	All
Logging Verbosity	Normal
Perform Code Analysis	AsConfigured
⊞ Source And Symbol Server Settinç	Index Sources
⊟ **2. Advanced**	
⊞ Agent Settings	Use agent where Name=* and Tags is e
Analyze Test Impact	True
Associate Changesets and Work I	True
Copy Outputs to Drop Folder	True
Create Work Item on Failure	True
Disable Tests	False
Get Version	
Label Sources	True
MSBuild Arguments	ployToTest\DeployToTest_20120228.3\
Private Drop Location	

MSBuild Arguments

Specify any additional command line arguments to pass to MSBuild.exe.

Note: Be sure to include a trailing slash at the end of the path to your build folder.

4. Click **Queue**.

When you queue the build, the project file will deploy the database scripts and web packages from the build drop folder you specified in the **OutputRoot** property.

Conclusion

This topic described how to publish deployment resources, like web packages and database scripts, from a specific previous build using the split project file deployment model. It explained how to override the **OutputRoot** property and how to incorporate the deployment logic into a TFS build definition.

Further Reading

For more information on creating build definitions, see Create a Basic Build Definition and Define Your Build Process. For more guidance on queuing builds, see Queue a Build.

Configuring Permissions for Team Build Deployment

This topic describes how to configure permissions to enable your build server to deploy content to web servers and database servers as part of an automated build process.

Task Overview

When you install the Team Foundation Server (TFS) 2010 build service, you specify the identity with which you want the service to run. By default, this is the Network Service account. Alternatively, you can configure the build service to run using a domain account.

Any deployment tasks that require Windows authentication, and that you plan to automate using Team Build, will run using the build service identity. As such, you'll need to grant the build service identity any required permissions on your web servers and your database servers.

> **Note:** The Network Service account uses the machine account to authenticate to other computers. Machine accounts take the form *[domain name]\[machine name]***$**—for example, **FABRIKAM\TFSBUILD$**. As such, if your build service runs using the Network Service identity, you should grant any required permissions to the machine account identity for your build server.

Configuring Web Server Permissions

As described in Choosing the Right Approach to Web Deployment, there are two main approaches you can use if you want to deploy web packages to a remote web server:

- Deploy the application from a remote location by targeting the *Web Deployment Agent Service* (also known as the remote agent) on the destination server.
- Deploy the application from a remote location by targeting the *Internet Information Services (IIS) Web Deploy Handler* on the destination server.

The remote agent has two key limitations in this case:

- The remote agent supports only NTLM authentication. In other words, the deployment must use the build service identity—you can't impersonate another account.
- To use the remote agent, the account that performs the deployment must be an administrator on the target server.

Together, these two limitations make the remote agent approach undesirable for an automated Team Build deployment. To use this approach, you'd need to make the build service account an administrator on any target web servers.

In contrast, the Web Deploy Handler approach offers various advantages:

- The Web Deploy Handler supports basic authentication over HTTPS, which allows you to pass the credentials of an alternative account to the IIS Web Deployment Tool (Web Deploy).
- You can configure target web servers to allow non-administrator users to deploy content to specific IIS websites using the Web Deploy Handler.

As a result, it's clearly preferable to target the Web Deploy Handler when you automate web package deployment from Team Build. This is the recommended process:

1. Create a low-privileged domain account to use for the deployment.
2. Configure the Web Deploy Handler and grant the account the required permissions to deploy content to a specific IIS website, as described in Configuring a Web Server for Web Deploy Publishing (Web Deploy Handler).
3. Invoke Web Deploy and target the Web Deploy Handler, using basic authentication and supplying the credentials of the domain account you created, to perform the deployment.

In the Contact Manager sample solution, you specify the authentication type (basic or NTLM), the Web Deploy credentials, and the endpoint address (remote agent or Web Deploy Handler) in the environment-specific project file. These values are used to formulate and run a Web Deploy command when the project file is executed. For more information, see Deploying Web Packages.

For more information on configuring the Web Deploy Handler, including how to configure permissions, see Configuring a Web Server for Web Deploy Publishing (Web Deploy Handler). For more information on configuring the remote agent, see Configuring a Web Server for Web Deploy Publishing (Remote Agent).

Configuring Database Server Permissions

To deploy a database to SQL Server, you must:

- Create a login for the deploying account on the SQL Server instance.
- Grant the login **DBCreator** permissions on the SQL Server instance.
- After the initial deployment, add the login to the **db_owner** role on the target database. This is required because on subsequent deployments, you're modifying an existing database rather than creating a new database.

You can authenticate to a SQL Server instance using either NTLM authentication or SQL Server authentication:

- If you use NTLM authentication, you need to grant the permissions described above to the build service account.

- If you use SQL Server authentication, you need to grant the permissions described above to the SQL Server account. You also need to include the SQL Server user name and password in the connection string you use to deploy the database.

For step-by-step details on how to configure permissions for database deployment, see Configuring a Database Server for Web Deploy Publishing.

Conclusion

At this point, you should understand the permissions required, together with the authentication options open to you, when you automate web application and database deployments from Team Build. You should also be able to implement the necessary permissions on IIS web servers and SQL Server database servers.

Further Reading

For more information on configuring Windows server environments to support remote deployment, see Configuring Server Environments for Web Deployment.

Advanced Enterprise Web Deployment

This tutorial will show you how to perform various tasks that are required or desirable in a lot of enterprise deployment scenarios.

Scenario Overview

The high-level scenario for these tutorials is described in Enterprise Web Deployment: Scenario Overview. We recommend that you review this topic before you get started on this tutorial.

How to Use This Tutorial

Each of the topics in this tutorial is self-contained and addresses a particular challenge or problem that occurs in enterprise deployment scenarios. You don't need to work through these topics in any particular order. However, this tutorial covers some advanced tasks. As such, you should familiarize yourself with the concepts and techniques that the Web Deployment in the Enterprise tutorial covers in order to gain the most benefit from this content.

This tutorial includes these topics:

- Performing a "What If" Deployment. In a lot of scenarios, you'll want to determine the impact of a proposed deployment on a target environment or any existing content before you actually make any changes. This topic describes how you can run a "what if" deployment to generate log files and database update scripts as if you had deployed content to a target environment, without actually making any changes. Analyzing these resources can help you to spot any potential problems in advance of a live deployment.
- Customizing Database Deployments for Multiple Environments. When you deploy a database project to multiple destinations, you'll often want to customize the deployment properties for each target environment. For example, in test environments you'd typically recreate the database on every deployment, whereas in staging or production environments you'd be a lot more likely to make incremental updates to preserve your data. This topic describes how you can incorporate these property changes into your deployment logic by creating an environment-specific deployment configuration (.sqldeployment) file for each target environment.
- Deploying Database Role Memberships to Test Environments. When you recreate a database on every deployment—for example, as part of a continuous integration (CI) build and deploy to a test environment—you'll typically need to configure database role memberships every time. For example, you'll usually need to grant permissions to the application pool identity associated with your web application. This topic describes how you can automate this process by adding a post-deployment SQL script to your deployment logic.
- Deploying Membership Databases to Enterprise Environments. ASP.NET membership databases have various characteristics that can complicate the deployment process. For example, a schema-only deployment will leave the database in a non-operational state. In most scenarios,

it's preferable to create a membership database directly in each destination environment. However, if you do have to deploy a membership database, this topic describes some of the approaches you can use to meet the inherent challenges.

- Excluding Files and Folders from Deployment. In some scenarios, you'll want to tailor the contents of your web package to specific destination environments. For example, you might want to include full versions of JavaScript libraries when you deploy to a test environment, to support client-side debugging, but use minified versions of the libraries when you deploy to a staging or production environment. This topic describes how you can exclude specific files and folders from the package creation process.
- Taking Web Applications Offline with Web Deploy. When you deploy solutions to a staging or production environment, you'll often want to take your web applications offline for the duration of the deployment process. This topic describes how you can add an *App_offline.htm* file to your web application at the start of the deployment process and remove it at the end. While the *App_offline.htm* file is in place, any users who browse to the web application are automatically redirected to the *App_offline.htm* file.
- Running Windows PowerShell Scripts from MSBuild. Many deployment scenarios require more complex post-deployment actions, like adding custom event sources to the registry or configuring replication between SQL Server instances. These actions are often accomplished through Windows PowerShell scripts. This topic describes how to run Windows PowerShell scripts from a Microsoft Build Engine (MSBuild) project file as part of the build and deployment process.
- Troubleshooting the Packaging Process. The Web Publishing Pipeline (WPP) defines an MSBuild property named **EnablePackageProcessLoggingAndAssert** that you can use to generate in-depth information about the packaging process for web application projects. This topic describes what the property does and how to use it.

Key Technologies

This tutorial focuses on how to use these products and technologies to support automated build and web deployment:

- Visual Studio 2010 and Team Foundation Server (TFS) 2010
- MSBuild and TFS Team Build
- Internet Information Services (IIS) 7.5
- IIS Web Deployment Tool (Web Deploy) 2.1
- The VSDBCMD.exe database deployment utility

Performing a "What If" Deployment

This topic describes how to perform "what if" (or simulated) deployments using the Internet Information Services (IIS) Web Deployment Tool (Web Deploy) and VSDBCMD. This lets you determine the effects of your deployment logic on a particular target environment before you actually deploy your application.

Performing a "What If" Deployment for Web Packages

Web Deploy includes functionality that lets you perform deployments in "what if" (or trial) mode. When you deploy artifacts in "what if" mode, Web Deploy generates a log file as if you had performed the deployment, but it doesn't actually change anything on the destination server. Reviewing the log file can help you to understand what impact your deployment will have on the destination server, in particular:

- What will get added.
- What will get updated.
- What will get deleted.

Because a "what if" deployment doesn't actually change anything on the destination server, what it can't always do is predict whether a deployment will succeed.

As described in Deploying Web Packages, you can deploy web packages using Web Deploy in two ways—by using the MSDeploy.exe command-line utility directly or by running the *.deploy.cmd* file that the build process generates.

If you're using MSDeploy.exe directly, you can run a "what if" deployment by adding the **–whatif** flag to your command. For example, to evaluate what would happen if you deployed the ContactManager.Mvc.zip package to a staging environment, the MSDeploy command should resemble this:

```
MSDeploy.exe
  -whatif
  -source:package="[path]\ContactManager.Mvc.zip"
  -dest:auto,
        computerName="https://stageweb1:8172/MSDeploy.axd?site=DemoSite",
        username="FABRIKAM\stagingdeployer",
        password="Pa$$w0rd",
        authtype="Basic",
        includeAcls="False"
  -verb:sync
  -disableLink:AppPoolExtension
  -disableLink:ContentExtension
  -disableLink:CertificateExtension
  -setParamFile:"[path]\ContactManager.Mvc.SetParameters.xml"
  -allowUntrusted
```

When you're satisfied with the results of your "what if" deployment, you can remove the **–whatif** flag to run a live deployment.

Note: For more information on command-line options for MSDeploy.exe, see Web Deploy Operation Settings.

If you're using the *.deploy.cmd* file, you can run a "what if" deployment by including the **/t** flag (trial mode) flag instead of the **/y** flag ("yes," or update mode) in your command. For example, to evaluate what would happen if you deployed the ContactManager.Mvc.zip package by running the *.deploy.cmd* file, your command should resemble this:

```
ContactManager.Mvc.deploy.cmd /t /m:TESTWEB1 /a:NTLM
```

When you're satisfied with the results of your "trial mode" deployment, you can replace the **/t** flag with a **/y** flag to run a live deployment:

```
ContactManager.Mvc.deploy.cmd /y /m:TESTWEB1 /a:NTLM
```

Note: For more information on command-line options for *.deploy.cmd* files, see How to: Install a Deployment Package Using the deploy.cmd File. If you run the *.deploy.cmd* file without specifying any flags, the command prompt will display a list of available flags.

Performing a "What If" Deployment for Databases

This section assumes that you're using the VSDBCMD utility to perform incremental, schema-based database deployment. This approach is described in more detail in Deploying Database Projects. We recommend that you familiarize yourself with this topic before you apply the concepts described here.

When you use VSDBCMD in **Deploy** mode, you can use the **/dd** (or **/DeployToDatabase**) flag to control whether VSDBCMD actually deploys the database or just generates a deployment script. If you're deploying a .dbschema file, this is the behavior:

- If you specify **/dd+** or **/dd**, VSDBCMD will generate a deployment script and deploy the database.
- If you specify **/dd-** or omit the switch, VSDBCMD will generate a deployment script only.

Note: If you're deploying a .deploymanifest file rather than a .dbschema file, the behavior of the **/dd** switch is a lot more complicated. Essentially, VSDBCMD will ignore the value of the **/dd** switch if the .deploymanifest file includes a **DeployToDatabase** element with a value of **True**. Deploying Database Projects describes this behavior in full.

For example, to generate a deployment script for the **ContactManager** database without actually deploying the database, your VSDBCMD command should resemble this:

```
vsdbcmd.exe /a:Deploy
            /manifest:"…\ContactManager.Database.deploymanifest"
            /cs:"Data Source=TESTDB1;Integrated Security=true"
            /p:TargetDatabase=ContactManager
            /dd-
            /script:"…\Publish-ContactManager-Db.sql"
```

VSDBCMD is a differential database deployment tool, and as such the deployment script is dynamically generated to contain all the SQL commands necessary to update the current database, if one exists, to the specified schema. Reviewing the deployment script is a useful way to determine what impact your deployment will have on the current database and the data it contains. For example, you might want to determine:

- Whether any existing tables will be removed, and whether that will result in data loss.
- Whether the order of operations carries a risk of data loss, for example, if you're splitting or merging tables.

If you're happy with the deployment script, you can repeat the VSDBCMD with a **/dd+** flag to make the changes. Alternatively, you can edit the deployment script to meet your requirements and then execute it manually on the database server.

Integrating "What If" Functionality into Custom Project Files

In more complex deployment scenarios, you'll want to use a custom Microsoft Build Engine (MSBuild) project file to encapsulate your build and deployment logic, as described in Understanding the Project File. For example, in the Contact Manager sample solution, the *Publish.proj* file:

- Builds the solution.
- Uses Web Deploy to package and deploy the ContactManager.Mvc application.
- Uses Web Deploy to package and deploy the ContactManager.Service application.
- Deploys the **ContactManager** database.

When you integrate the deployment of multiple web packages and/or databases into a single-step process in this way, you may also want the option of performing the entire deployment in a "what if" mode.

The *Publish.proj* file demonstrates how you can do this. First, you need to create a property to store the "what if" value:

XML

```xml
<PropertyGroup>
  <WhatIf Condition=" '$(WhatIf)'=='' ">false</WhatIf>
</PropertyGroup>
```

In this case, you've created a property named **WhatIf** with a default value of **false**. Users can override this value by setting the property to **true** in a command-line parameter, as you'll see shortly.

The next stage is to parameterize any Web Deploy and VSDBCMD commands so that the flags reflect the **WhatIf** property value. For example, the next target (taken from the *Publish.proj* file and simplified) runs the *.deploy.cmd* file to deploy a web package. By default, the command includes a **/Y** switch ("yes," or update mode). If **WhatIf** is set to **true**, this is replaced by a **/T** switch (trial, or "what if" mode).

XML

```xml
<Target Name="PublishWebPackages" Outputs="%(PublishPackages.Identity)">
  <PropertyGroup>
    <_WhatIfSwitch>/Y</_WhatIfSwitch>
    <_WhatIfSwitch Condition=" '$(WhatIf)'=='true' ">/T</_WhatIfSwitch>
    <_Cmd>%(PublishPackages.FullPath) $(_WhatifSwitch)
         /M:$(MSDeployComputerName)
         /U:$(MSDeployUsername)
         /P:$(MSDeployPassword)
         /A:$(MSDeployAuth)
         %(PublishPackages.AdditionalMSDeployParameters)
    </_Cmd>
  </PropertyGroup>
  <Exec Command="$(_Cmd)"/>
</Target>
```

Similarly, the next target uses the VSDBCMD utility to deploy a database. By default, a **/dd** switch is not included. This means that VSDBCMD will generate a deployment script but will not deploy the database—in other words, a "what if" scenario. If the **WhatIf** property is not set to **true**, a **/dd** switch is added and VSDBCMD will deploy the database.

XML

```xml
<Target Name="PublishDbPackages" Outputs="%(DbPublishPackages.Identity)">
  <PropertyGroup>
    <_DbDeployOrScript></_DbDeployOrScript>
    <_DbDeployOrScript Condition=" '$(Whatif)'!='true' ">/dd</_DbDeployOrScript>
    <_Cmd>"$(VsdbCmdExe)" /a:Deploy
           /cs:"%(DbPublishPackages.DatabaseConnectionString)"
           /p:TargetDatabase=%(DbPublishPackages.TargetDatabase)
           /manifest:"%(DbPublishPackages.FullPath)"
           /script:"$(_CmDbScriptPath)"
           $(_DbDeployOrScript)
    </_Cmd>
  </PropertyGroup>
  <Exec Command="$(_Cmd)"/>
</Target>
```

You can use the same approach to parameterize all the relevant commands in your project file. When you want to run a "what if" deployment, you can then simply provide a **WhatIf** property value from the command line:

```
MSBuild.exe Publish.proj /p:WhatIf=true;TargetEnvPropsFile=EnvConfig\Env-Dev.proj
```

In this way, you can run a "what if" deployment for all your project components in a single step.

Conclusion

This topic described how to run "what if" deployments using Web Deploy, VSDBCMD, and MSBuild. A "what if" deployment lets you evaluate the impact of a proposed deployment before you actually make any changes to the destination environment.

Further Reading

For more information on Web Deploy command-line syntax, see Web Deploy Operation Settings. For guidance on command-line options when you use the *.deploy.cmd* file, see How to: Install a Deployment Package Using the deploy.cmd File. For guidance on VSDBCMD command-line syntax, see Command-Line Reference for VSDBCMD.EXE (Deployment and Schema Import).

Customizing Database Deployments for Multiple Environments

This topic describes how to tailor the properties of a database to specific target environments as part of the deployment process.

> **Note:** The topic assumes that you're deploying a Visual Studio 2010 database project using MSBuild.exe and VSDBCMD.exe. For more information on why you might choose this approach, see Web Deployment in the Enterprise and Deploying Database Projects.

When you deploy a database project to multiple destinations, you'll often want to customize the database deployment properties for each target environment. For example, in test environments you'd typically recreate the database on every deployment, whereas in staging or production environments you'd be a lot more likely to make incremental updates to preserve your data.

In a Visual Studio 2010 database project, deployment settings are contained within a deployment configuration (.sqldeployment) file. This topic will show you how to create environment-specific deployment configuration files and specify the one you want to use as a VSDBCMD parameter.

Task Overview

This topic assumes that:

- You use the split project file approach to solution deployment, as described in Understanding the Project File.
- You call VSDBCMD from the project file to deploy your database project, as described in Understanding the Build Process.

To create a deployment system that supports varying the database deployment properties between target environments, you'll need to:

- Create a deployment configuration (.sqldeployment) file for each target environment.
- Create a VSDBCMD command that specifies the deployment configuration file as a command-line switch.

- Parameterize the VSDBCMD command in a Microsoft Build Engine (MSBuild) project file, so that the VSDBCMD options are appropriate to the target environment.

This topic will show you how to perform each of these procedures.

Creating Environment-Specific Deployment Configuration Files

By default, a database project contains a single deployment configuration file named *Database.sqldeployment*. If you open this file in Visual Studio 2010, you can see the different deployment options that are available to you:

- **Deployment comparison collation**. This lets you choose whether to use the database collation of your project (the *source* collation) or the database collation of your destination server (the *target* collation). In most cases, you'll want to use the source collation when you deploy to a development or test environment. When you deploy to a staging or production environment, you'll usually want to leave the target collation unchanged to avoid any interoperability issues.

- **Deploy database properties**. This lets you choose whether to apply the database properties, as defined in the *Database.sqlsettings* file. When you deploy a database for the first time, you should deploy the database properties. If you're updating an existing database, the properties should already be in place, and you shouldn't need to deploy them again.

- **Always re-create database**. This lets you choose whether to re-create the target database every time you deploy or make incremental changes to bring the target database up to date with your schema. If you re-create the database, you'll lose any data in the existing database. As such, you should usually set this to **false** for deployments to staging or production environments.

- **Block incremental deployment if data loss might occur**. This lets you choose whether deployment should stop if a change to the database schema will cause the loss of data. You typically set this to **true** for a deployment to a production environment, to give you the opportunity to intervene and protect any important data. If you have set **Always re-create database** to **false**, this setting will have no effect.

- **Execute deployment in single-user mode**. This is not usually an issue in development or test environments. However, you should typically set this to **true** for deployments to staging or production environments. This prevents users from making changes to the database while the deployment is underway.

- **Back up database before deployment**. You typically set this to **true** when you deploy to a production environment, as a precaution against data loss. You may also want to set it to **true** when you deploy to a staging environment, if your staging database contains a lot of data.

- **Generate DROP statements for objects that are in the target database but that are not in the database project**. In most cases, this is an integral and essential part of making incremental changes to a database. If you have set **Always re-create database** to **false**, this setting will have no effect.

- **Do not use ALTER ASSEMBLY statements to update CLR types**. This setting determines how SQL Server should update common language runtime (CLR) types to newer assembly versions. This should be set to **false** in most scenarios.

This table shows typical deployment settings for different destination environments. However, your settings may be different depending on your exact requirements.

	Developer/Test	Staging/Integration	Production
Deployment comparison collation	Source	Target	Target
Deploy database properties	True	First time only	First time only
Always re-create database	True	False	False
Block incremental deployment if data loss might occur	False	Maybe	True
Execute deployment script in single-user mode	False	True	True
Back up database before deployment	False	Maybe	True
Generate DROP statements for objects that are in the target database but that are not in the database project	False	True	True
Do not use ALTER ASSEMBLY statements to update CLR types	False	False	False

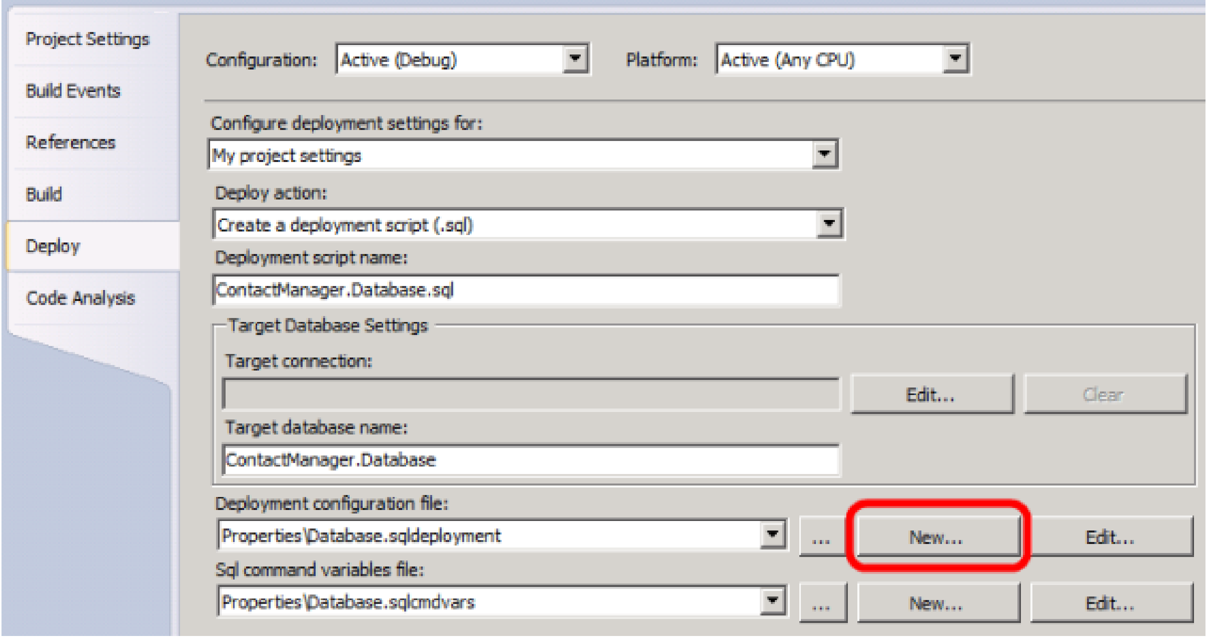
Note: For more information on database deployment properties and environment considerations, see An Overview of Database Project Settings, How to: Configure Properties for Deployment Details, Build and Deploy Database to an Isolated Development Environment, and Build and Deploy Databases to a Staging or Production Environment.

To support the deployment of a database project to multiple destinations, you should create a deployment configuration file for each target environment.

To create an environment-specific configuration file

1. In Visual Studio 2010, in the **Solution Explorer** window, right-click your database project, and then click **Properties**.

2. On the database project properties page, on the **Deploy** tab, in the **Deployment configuration file** row, click **New**.



Configuration: Active (Debug)

Platform: Active (Any CPU)

Configure deployment settings for: My project settings

Deploy action: Create a deployment script (.sql)

Deployment script name: ContactManager.Database.sql

Target Database Settings

Target connection:

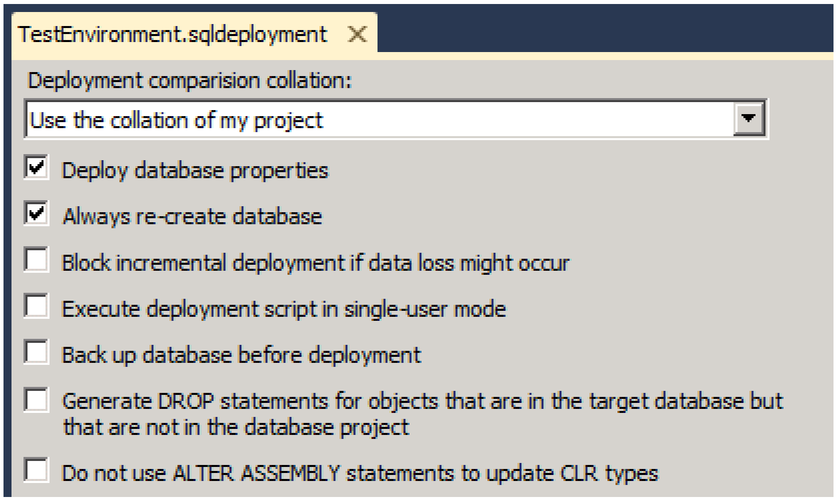
Target database name: ContactManager.Database

Deployment configuration file: Properties\Database.sqldeployment

Sql command variables file: Properties\Database.sqlcmdvars

3. In the **New Deployment Configuration File** dialog box, give the file a meaningful name (for example, **TestEnvironment.sqldeployment**), and then click **Save**.

4. On the *[Filename]***.sqldeployment** page, set the deployment properties to match the requirements of your destination environment, and then save the file.



Deployment comparision collation: Use the collation of my project

☑ Deploy database properties

☑ Always re-create database

□ Block incremental deployment if data loss might occur

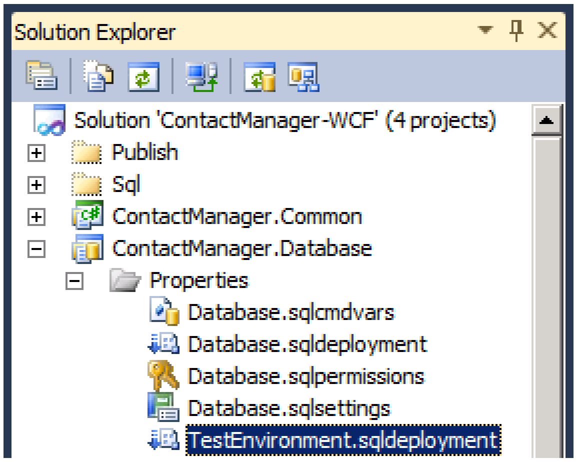
□ Execute deployment script in single-user mode

□ Back up database before deployment

□ Generate DROP statements for objects that are in the target database but that are not in the database project

□ Do not use ALTER ASSEMBLY statements to update CLR types

5. Notice that the new file is added to the Properties folder in your database project.



Solution Explorer

Solution 'ContactManager-WCF' (4 projects)
- Publish
- Sql
- ContactManager.Common
- ContactManager.Database
  - Properties
    - Database.sqlcmdvars
    - Database.sqldeployment
    - Database.sqlpermissions
    - Database.sqlsettings
    - TestEnvironment.sqldeployment

Specifying the Deployment Configuration File in VSDBCMD

When you use solution configurations (like Debug and Release) within Visual Studio 2010, you can associate a deployment configuration file with each configuration. When you build a particular configuration, the build process generates a configuration-specific deployment manifest file that points to the configuration-specific deployment configuration file. However, one of the main aims of the approach to deployment described in these tutorials is to give people the ability to control the deployment process without using Visual Studio 2010 and solution configurations. In this approach, the solution configuration is the same regardless of the target deployment environment. To tailor your database deployment to a specific destination environment, you can use the VSDBCMD command-line options to specify your deployment configuration file.

To specify a deployment configuration file in your VSDBCMD, use the **p:/DeploymentConfigurationFile** switch and provide the full path to your file. This will override the deployment configuration file that the deployment manifest identifies. For example, you could use this VSDBCMD command to deploy the **ContactManager** database to a test environment:

```
vsdbcmd.exe /a:Deploy
            /manifest:"…\ContactManager.Database.deploymanifest"
            /cs:"Data Source=TESTDB1;Integrated Security=true"
            /p:TargetDatabase=ContactManager
            /p:DeploymentConfigurationFile=
              "…\ContactManager.Database_TestEnvironment.sqldeployment"
            /dd+
            /script:"…\Publish-ContactManager-Db.sql"
```

Note: Note that the build process may rename your .sqldeployment file when it copies the file to the output directory.

If you use SQL command variables in your pre-deployment or post-deployment SQL scripts, you can use a similar approach to associate an environment-specific .sqlcmdvars file with your deployment. In this case, you use the **p:/SqlCommandVariablesFile** switch to identify your .sqlcmdvars file.

Running the VSDBCMD Command from an MSBuild Project File

You can invoke a VSDBCMD command from an MSBuild project file by using an **Exec** task within an MSBuild target. In its simplest form, it would look like this:

XML

```
<Target Name="DeployDatabase">
   <PropertyGroup>
      <_Cmd>
         Add your VSDBCMD command here
      </_Cmd>
   </PropertyGroup>
   <Exec Command="$(_Cmd)"/>
 </Target>
```

In practice, to make your project files easy to read and reuse, you'll want to create properties to store the various command-line parameters. This makes it easier for users to provide property values in an environment-specific project file or to override default values from the MSBuild command line. If you use the split project file approach described in Understanding the Project File, you should divide your build instructions and properties between the two files accordingly:

- Environment-specific settings, like the deployment configuration filename, the database connection string, and the target database name, should go in the environment-specific project file.
- The MSBuild target that runs the VSDBCMD command, together with any universal properties like the location of the VSDBCMD executable, should go in the universal project file.

You should also ensure that you build the database project before you invoke VSDBCMD so that the .deploymanifest file is created and ready to use. You can see a full example of this approach in the topic Understanding the Build Process, which walks you through the project files in the Contact Manager sample solution.

Conclusion

This topic described how you can tailor database properties to different destination environments when you deploy database projects using MSBuild and VSDBCMD. This approach is useful when you need to deploy database projects as part of larger, enterprise-scale solutions. These solutions are often deployed to multiple destinations, like sandboxed development or test environments, staging or integration platforms, and production or live environments. Each of these target environments typically requires a unique set of database deployment properties.

Further Reading

For more information on deploying database projects using VSDBCMD.exe, see Deploying Database Projects. For more information on using custom MSBuild project files to control the deployment process, see Understanding the Project File and Understanding the Build Process.

These articles on MSDN provide more general guidance on database deployment:

- An Overview of Database Project Settings
- How to: Configure Properties for Deployment Details
- Build and Deploy Databases to an Isolated Development Environment
- Build and Deploy Databases to a Staging or Production Environment

Deploying Database Role Memberships to Test Environments

This topic describes how to add user accounts to database roles as part of a solution deployment to a test environment.

When you deploy a solution containing a database project to a staging or production environment, you typically don't want the developer to automate the addition of user accounts to database roles. In most cases, the developer won't know which user accounts need to be added to which database roles, and these requirements could change at any time. However, when you deploy a solution containing a database project to a development or test environment, the situation is usually rather different:

- The developer typically re-deploys the solution on a regular basis, often several times a day.
- The database is typically re-created on every deployment, which means that database users must be created and added to roles after every deployment.
- The developer typically has full control over the target development or test environment.

In this scenario, it's often beneficial to automatically create database users and assign database role memberships as part of the deployment process.

The key factor is that this operation needs to be conditional based on the target environment. If you're deploying to a staging or a production environment, you want to skip the operation. If you're deploying to a developer or test environment, you want to deploy role memberships without further intervention. This topic describes one approach you can use to address this challenge.

Task Overview

This topic assumes that:

- You use the split project file approach to solution deployment, as described in Understanding the Project File.
- You call VSDBCMD from the project file to deploy your database project, as described in Understanding the Build Process.

To create database users and assign role memberships when you deploy a database project to a test environment, you'll need to:

- Create a Transact Structured Query Language (Transact-SQL) script that makes the necessary database changes.
- Create a Microsoft Build Engine (MSBuild) target that uses the sqlcmd.exe utility to run the SQL script.
- Configure your project files to invoke the target when you're deploying your solution to a test environment.

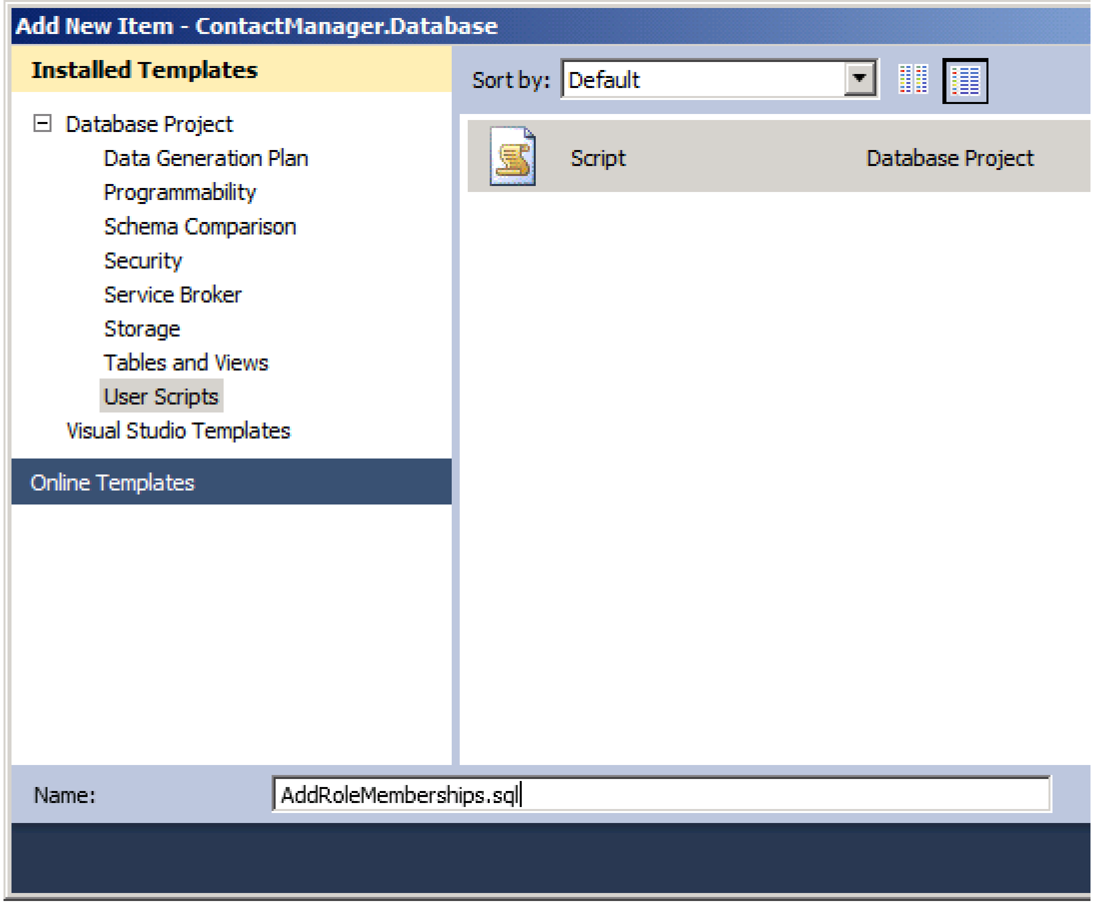
This topic will show you how to perform each of these procedures.

Scripting the Database Role Memberships

You can create a Transact-SQL script in a lot of different ways, and in any location you choose. The easiest approach is to create the script within your solution in Visual Studio 2010.

To create a SQL script

1. In the **Solution Explorer** window, expand your database project node.
2. Right-click the **Scripts** folder, point to **Add**, and then click **New Folder**.
3. Type **Test** as the folder name, and then press Enter.
4. Right-click the **Test** folder, point to **Add**, and then click **Script**.
5. In the **Add New Item** dialog box, give your script a meaningful name (for example, **AddRoleMemberships.sql**), and then click **Add**.



Name: AddRoleMemberships.sql

6. In the *AddRoleMemberships.sql* file, add Transact-SQL statements that:

   a. Create a database user for the SQL Server login that will access your database.

   b. Add the database user to any required database roles.

   The file should resemble this:

Transact-SQL

```
USE $(DatabaseName)
GO
CREATE USER [FABRIKAM\TESTWEB1$] FOR LOGIN[FABRIKAM\TESTWEB1$]
GO
USE [ContactManager]
GO
EXEC sp_addrolemember N'db_datareader', N'FABRIKAM\TESTWEB1$'
GO
USE [ContactManager]
GO
EXEC sp_addrolemember N'db_datawriter', N'FABRIKAM\TESTWEB1$'
GO
```

7. Save the file.

Executing the Script on the Target Database

Ideally, you'd run any required Transact-SQL scripts as part of a post-deployment script when you deploy your database project. However, post-deployment scripts don't allow you to execute logic conditionally based on solution configurations or build properties. The alternative is to run your SQL scripts directly from the MSBuild project file, by creating a **Target** element that executes a sqlcmd.exe command. You can use this command to run your script on the target database:

```
sqlcmd.exe –S [Database server] –d [Database name] –i [SQL script]
```

Note: For more information on sqlcmd command-line options, see sqlcmd Utility.

Before you embed this command in an MSBuild target, you need to consider under what conditions you want the script to run:

- The target database must exist before you change its role memberships. As such, you need to run this script *after* the database deployment.
- You need to include a condition so that the script is only executed for test environments.
- If you're running a "what if" deployment—in other words, if you're generating deployment scripts but not actually running them—you shouldn't run the SQL script.

If you're using the split project file approach described in Understanding the Project File, as demonstrated by the Contact Manager sample solution, you can split the build instructions for your SQL script like this:

- Any required environment-specific properties, together with the property that determines whether to deploy permissions, should go in the environment-specific project file (for example, *Env-Dev.proj*).
- The MSBuild target itself, together with any properties that will not change between destination environments, should go in the universal project file (for example, *Publish.proj*).

In the environment-specific project file, you need to define the database server name, the target database name, and a Boolean property that lets the user specify whether to deploy role memberships.

XML

```
<PropertyGroup>
   <CmTargetDatabase Condition=" '$(CmTargetDatabase)'=='' ">
      ContactManager
   </CmTargetDatabase>
   <DatabaseServer Condition=" '$(DatabaseServer)'=='' ">
      TESTDB1
   </DatabaseServer>
   <DeployTestDBRoleMemberships Condition="'$(DeployTestDBRoleMemberships)'==''">
      true
   </DeployTestDBRoleMemberships>
</PropertyGroup>
```

In the universal project file, you need to provide the location of the sqlcmd executable and the location of the SQL script you want to run. These properties will remain the same regardless of the destination environment. You also need to create an MSBuild target to execute the sqlcmd command.

XML

```
<PropertyGroup>
   <SqlCmdExe Condition=" '$(SqlCmdExe)'=='' ">
      C:\Program Files\Microsoft SQL Server\100\Tools\Binn\sqlcmd.exe
   </SqlCmdExe>
</PropertyGroup>

<Target Name="DeployTestDBPermissions"
        Condition=" '$(DeployTestDBRoleMemberships)'=='true' AND
                    '$(Whatif)'!='true' ">
   <PropertyGroup>
     <SqlScript>
        $(SourceRoot)ContactManager.Database\Scripts\Test\AddRoleMemberships.sql
     </SqlScript>
     <_Cmd>"$(SqlCmdExe)" -S "$(DatabaseServer)"
                          -d "$(CmTargetDatabase)"
                          -i "$(SqlScript)"
     </_Cmd>
   </PropertyGroup>
   <Exec Command="$(_Cmd)" ContinueOnError="false" />
</Target>
```

Notice that you add the location of the sqlcmd executable as a static property, as this could be useful to other targets. In contrast, you define the location of your SQL script and the syntax of the sqlcmd command as dynamic properties within the target, as they will not be required before the target is executed. In this case, the **DeployTestDBPermissions** target will only be executed if these conditions are met:

- The **DeployTestDBRoleMemberships** property is set to **true**.
- The user hasn't specified a **WhatIf=true** flag.

Finally, don't forget to invoke the target. In the *Publish.proj* file, you can do this by adding the target to the dependency list for the default **FullPublish** target. You need to ensure that the **DeployTestDBPermissions** target is not executed until the **PublishDbPackages** target has been executed.

XML

```
<Project ToolsVersion="4.0"
         DefaultTargets="FullPublish"
         xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
```

```
   ...
   <PropertyGroup>
      <FullPublishDependsOn>
         Clean;
         BuildProjects;
         GatherPackagesForPublishing;
         PublishDbPackages;
         DeployTestDBPermissions;
         PublishWebPackages;
      </FullPublishDependsOn>
   </PropertyGroup>
   <Target Name="FullPublish" DependsOnTargets="$(FullPublishDependsOn)" />
</Project>
```

Conclusion

This topic described one way in which you can add database users and role memberships as a post-deployment action when you deploy a database project. This is typically useful when you regularly re-create a database in a test environment, but it should usually be avoided when you deploy databases to staging or production environments. As such, you should ensure that you use the necessary conditional logic so that database users and role memberships are only created when it's appropriate to do so.

Further Reading

For more information on using VSDBCMD to deploy database projects, see Deploying Database Projects. For guidance on customizing database deployments for different target environments, see Customizing Database Deployments for Multiple Environments. For more information on using custom MSBuild project files to control the deployment process, see Understanding the Project File and Understanding the Build Process. For more information on sqlcmd command-line options, see sqlcmd Utility.

Deploying Membership Databases to Enterprise Environments

This topic explains the key considerations and challenges you'll need to overcome when you provision ASP.NET application services databases (more commonly referred to as membership databases) in test, staging, or production environments. It also describes approaches you can use to meet these challenges.

What Are the Issues When You Deploy a Membership Database?

In most cases, when you devise a deployment strategy for a database, the first thing you need to consider is what data you want to deploy. In a development or test environment, you might want to deploy user account data to facilitate quick and easy testing. In a staging or production environment, it's very unlikely that you'd want to deploy user account data.

Unfortunately, ASP.NET membership databases introduce some specific challenges that make this decision a lot more complex:

- A schema-only deployment will leave the membership database in a non-operational state. This is because the membership database includes some configuration data (in the

aspnet_SchemaVersions table) that the database requires in order to function. As such, if you perform a schema-only deployment of your membership database in order to exclude user account data, you'll need to run a post-deployment script to add the essential configuration data.

- Depending on how your membership database is configured, the membership provider may use the machine key to encrypt passwords and store them in the database. In this case, any user account data you deploy with the database will become unusable on the destination server. For this reason, deploying user account data is not a supported scenario.

Choosing a Membership Database Strategy

Use these guidelines when you choose how to provision a membership database in an enterprise server environment:

- Wherever possible, do not deploy membership databases. Instead, create the membership database manually on the target database server. If you haven't customized your membership database schema, you can simply create a new one in situ at the destination using the ASP.NET SQL Server Registration Tool (aspnet_regsql.exe).
- If you have no option but to deploy a membership database—for example, if you've made extensive modifications to the database schema—you should perform a schema-only deployment of the membership database, to exclude user account data, and then run a post-deployment script to add any required configuration data. You can find broad guidance on these approaches in How to: Deploy the ASP.NET Membership Database Without Including User Accounts.

It's important to remember that *the schema of your membership database is likely to be fairly static*. Even if you've customized the membership database, it's unlikely that you'll need to update the schema on a regular basis—it's not going to change with the same frequency as the code in a web application or a database project. As such, you shouldn't need to include the membership database in any automated or single-step deployment processes.

Using VSDBCMD to Update a Membership Database Schema

If you modify the structure of your membership database after your first deployment, you may not want to use the Internet Information Services (IIS) Web Deployment Tool (Web Deploy) to redeploy the database. The database deployment functionality in Web Deploy doesn't include the capability to make differential updates to a destination database—instead, Web Deploy must drop and re-create the database. This means that you lose any existing user account data, which is typically undesirable in staging or production environments.

The alternative is to use the VSDBCMD utility to update the schema of your destination database. VSDBCMD includes two important capabilities. First, it can import the schema of an existing database into a .dbschema file. Second, it can deploy a .dbschema file to an existing database as a differential update, which means that it only makes the changes required to bring the target database up to date and you don't lose any data.

You can use these high-level steps to update a membership database schema:

1. Use the VSDBCMD **Import** action to generate a .dbschema file for your source membership database. This procedure is described in How to: Import a Schema from a Command Prompt.
2. Use the VSDBCMD **Deploy** action to deploy the .dbschema file to your destination membership database. This procedure is described in Command-Line Reference for VSDBCMD.EXE (Deployment and Schema Import).

Conclusion

This topic described some of the challenges you may face when you need to provision ASP.NET membership databases in various target environments. In particular, it explained why schema-only deployments will leave the membership database in a non-operational state and why deploying user account data is not supported. The topic also presented guidance on how to provision, deploy, and update membership databases in different scenarios.

Further Reading

For more guidance and examples of how to use VSDBCMD, see Command-Line Reference for VSDBCMD.EXE (Deployment and Schema Import) and How to: Import a Schema from a Command Prompt. For more information on using aspnet_regsql.exe to create membership databases, see ASP.NET SQL Server Registration Tool (aspnet_regsql.exe). For more general guidance on deploying membership databases, see How to: Deploy the ASP.NET Membership Database Without Including User Accounts.

Excluding Files and Folders from Deployment

This topic describes how you can exclude files and folders from a web deployment package when you build and package a web application project.

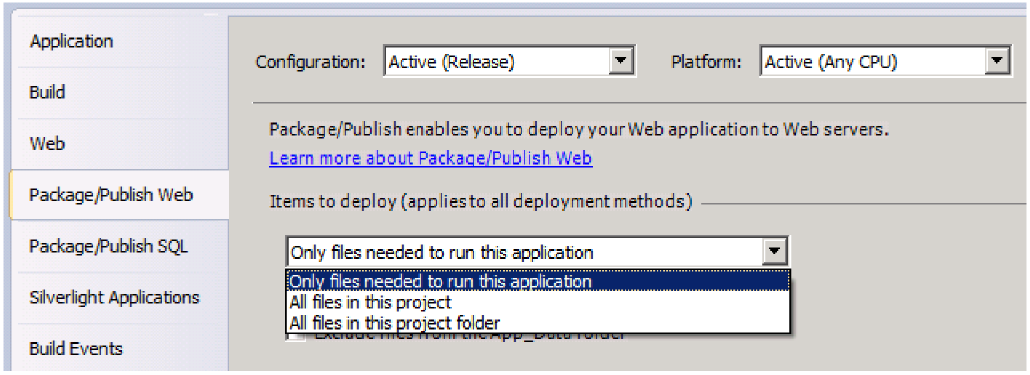
Overview

When you build a web application project in Visual Studio 2010, the Web Publishing Pipeline (WPP) lets you extend this build process by packaging your compiled web application into a deployable web package. You can then use the Internet Information Services (IIS) Web Deployment Tool (Web Deploy) to deploy this web package to a remote IIS web server, or import the web package manually through IIS Manager. This packaging process is explained in Building and Packaging Web Application Projects.

So how do you control what gets included in your web package? The project settings in Visual Studio, through the underlying project file, provide sufficient control for a lot of scenarios. However, in some cases you may want to tailor the contents of your web package to specific destination environments. For example, you might want to include a folder for log files when you deploy your application to a test environment but exclude the folder when you deploy the application to a staging or production environment. This topic will show you how to do this.

What Gets Included by Default?

When you configure your web application project properties in Visual Studio, the **Items to deploy** list on the **Package/Publish Web** page lets you specify what you want to include in your web deployment package. By default, this is set to **Only files needed to run this application**.



Configuration: Active (Release)

Platform: Active (Any CPU)

Package/Publish enables you to deploy your Web application to Web servers.

Learn more about Package/Publish Web

Items to deploy (applies to all deployment methods)

Only files needed to run this application

- Only files needed to run this application
- All files in this project
- All files in this project folder

Exclude files from the App_Data folder

When you choose **Only files needed to run this application**, the WPP will try to determine which files should be added to the web package. This includes:

- All the build outputs for the project.
- Any files marked with a build action of **Content**.

Note: The logic that determines which files to include is contained in this file:

%PROGRAMFILES%\MSBuild\Microsoft\VisualStudio\v10.0\Web\ Microsoft.Web.Publishing.OnlyFilesToRunTheApp.targets

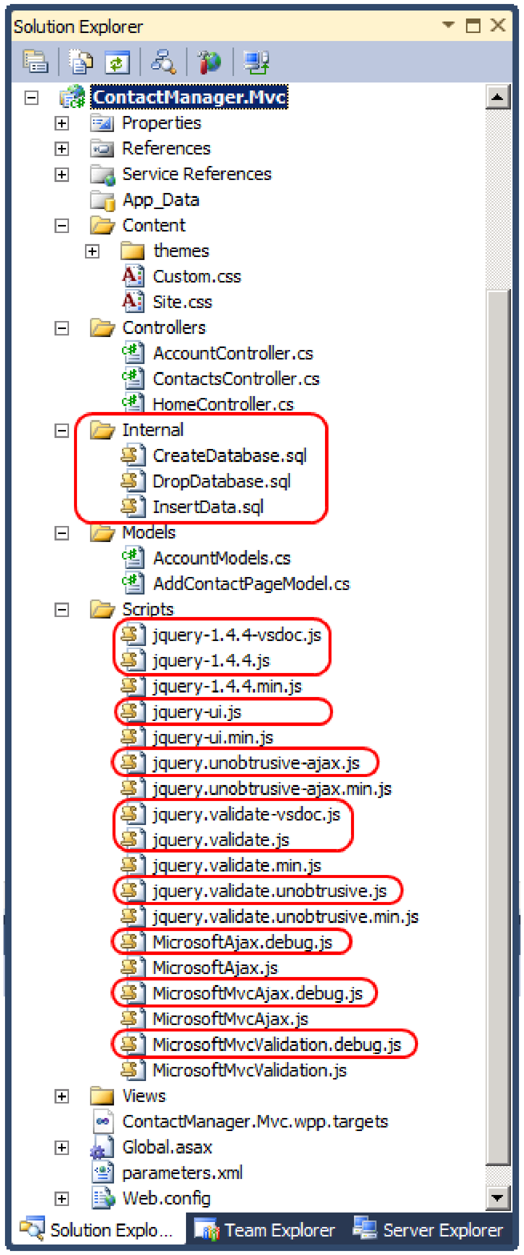
Excluding Specific Files and Folders

In some cases, you'll want more fine-grained control over which files and folders are deployed. If you know which files you want to exclude ahead of time, and the exclusion applies to all destination environments, you can simply set the **Build Action** of each file to **None**.

To exclude specific files from deployment

1. In the **Solution Explorer** window, right-click the file, and then click **Properties**.
2. In the **Properties** window, in the **Build Action** row, select **None**.

However, this approach is not always convenient. For example, you may want to vary which files and folders are included according to your destination environment, and from outside Visual Studio. For example, in the Contact Manager sample solution, take a look at the contents of the ContactManager.Mvc project:



```
ContactManager.Mvc
    Properties
    References
    Service References
    App_Data
    Content
        themes
        Custom.css
        Site.css
    Controllers
        AccountController.cs
        ContactsController.cs
        HomeController.cs
    Internal
        CreateDatabase.sql
        DropDatabase.sql
        InsertData.sql
    Models
        AccountModels.cs
        AddContactPageModel.cs
    Scripts
        jquery-1.4.4-vsdoc.js
        jquery-1.4.4.js
        jquery-1.4.4.min.js
        jquery-ui.js
        jquery-ui.min.js
        jquery.unobtrusive-ajax.js
        jquery.unobtrusive-ajax.min.js
        jquery.validate-vsdoc.js
        jquery.validate.js
        jquery.validate.min.js
        jquery.validate.unobtrusive.js
        jquery.validate.unobtrusive.min.js
        MicrosoftAjax.debug.js
        MicrosoftAjax.js
        MicrosoftMvcAjax.debug.js
        MicrosoftMvcAjax.js
        MicrosoftMvcValidation.debug.js
        MicrosoftMvcValidation.js
    Views
    ContactManager.Mvc.wpp.targets
    Global.asax
    parameters.xml
    Web.config
```

- The Internal folder contains some SQL scripts that the developer uses to create, drop, and populate local databases for development purposes. Nothing in this folder should be deployed to a staging or production environment.
- The Scripts folder contains several JavaScript files. A lot of these files are included purely to support debugging or provide IntelliSense in Visual Studio. Some of these files should not be deployed to staging or production environments. However, you may want to deploy them to a developer test environment to facilitate troubleshooting.

Although you could manipulate your project files to exclude specific files and folders, there is an easier way. The WPP includes a mechanism to exclude files and folders by building item lists named **ExcludeFromPackageFolders** and **ExcludeFromPackageFiles**. You can extend this mechanism by adding your own items to these lists. To do this, you need to complete these high-level steps:

1. Create a custom project file named *[project name].wpp.targets* in the same folder as your project file.

   > **Note:** The *.wpp.targets* file needs to go in the same folder as your web application project file—for example, *ContactManager.Mvc.csproj*—rather than in the same folder as any custom project files you use to control the build and deployment process.

2. In the *.wpp.targets* file, add an **ItemGroup** element.
3. In the **ItemGroup** element, add **ExcludeFromPackageFolders** and **ExcludeFromPackageFiles** items to exclude specific files and folders as required.

This is the basic structure of this *.wpp.targets* file:

XML

```xml
<Project ToolsVersion="4.0"
         xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <ItemGroup>
    <ExcludeFromPackageFolders Include="[semi-colon-separated folder list]">
      <FromTarget>[arbitrary metadata value]</FromTarget>
    </ExcludeFromPackageFolders>
    <ExcludeFromPackageFiles Include="[semi-colon-separated file list]">
      <FromTarget>[arbitrary metadata value]</FromTarget>
    </ExcludeFromPackageFiles>
  </ItemGroup>
</Project>
```

Note that each item includes an item metadata element named **FromTarget**. This is an optional value that doesn't affect the build process; it simply serves to indicate why particular files or folders were omitted if someone reviews the build logs.

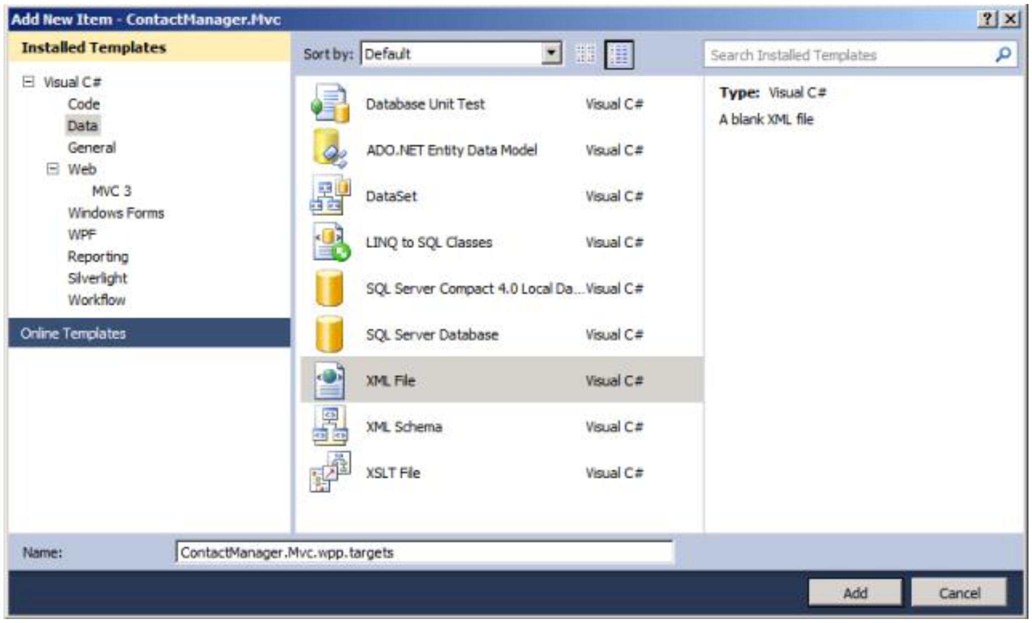
Excluding Files and Folders from a Web Package

The next procedure shows you how to add a *.wpp.targets* file to a web application project and how to use the file to exclude specific files and folders from the web package when you build your project.

To exclude files and folders from a web deployment package

1. Open your solution in Visual Studio 2010.
2. In the **Solution Explorer** window, right-click your web application project node (for example, **ContactManager.Mvc**), point to **Add**, and then click **New Item**.
3. In the **Add New Item** dialog box, select the **XML File** template.

4. In the **Name** box, type *[project name]***.wpp.targets** (for example, **ContactManager.Mvc.wpp.targets**), and then click **Add**.



Sort by: Default

Name: ContactManager.Mvc.wpp.targets

Note: If you add a new item to the root node of a project, the file is created in the same folder as the project file. You can verify this by opening the folder in Windows Explorer.

5. In the file, add a **Project** element and an **ItemGroup** element:

XML

```xml
<Project ToolsVersion="4.0"
         xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <ItemGroup>
  </ItemGroup>
</Project>
```

6. If you want to exclude folders from the web package, add an **ExcludeFromPackageFolders** element to the **ItemGroup** element:

    a. In the **Include** attribute, provide a semicolon-separated list of the folders you want to exclude.

    b. In the **FromTarget** metadata element, provide a meaningful value to indicate why the folders are being excluded, like the name of the *.wpp.targets* file.

XML

```xml
<ExcludeFromPackageFolders Include="Internal">
  <FromTarget>ContactManager.Mvc.wpp.targets</FromTarget>
</ExcludeFromPackageFolders>
```

7. If you want to exclude files from the web package, add an **ExcludeFromPackageFiles** element to the **ItemGroup** element:

a. In the **Include** attribute, provide a semicolon-separated list of the files you want to exclude.

b. In the **FromTarget** metadata element, provide a meaningful value to indicate why the files are being excluded, like the name of the *.wpp.targets* file.

XML

```xml
<ExcludeFromPackageFiles Include="Scripts\jquery-1.4.4-
vsdoc.js;Scripts\jquery-1.4.4.js;Scripts\jquery-
ui.js;Scripts\jquery.unobtrusive-ajax.js;Scripts\jquery.validate-
vsdoc.js;Scripts\jquery.validate.js;Scripts\jquery.validate.unobtrusive.js;Scr
ipts\MicrosoftAjax.debug.js;Scripts\MicrosoftMvcValidation.debug.js">
  <FromTarget>ContactManager.Mvc.wpp.targets</FromTarget>
</ExcludeFromPackageFiles>
```

8. The *[project name].wpp.targets* file should now resemble this:

XML

```xml
<Project ToolsVersion="4.0"
         xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <ItemGroup>
    <ExcludeFromPackageFolders Include="Internal">
      <FromTarget>ContactManager.Mvc.wpp.targets</FromTarget>
    </ExcludeFromPackageFolders>
    <ExcludeFromPackageFiles Include="Scripts\jquery-1.4.4-
vsdoc.js;Scripts\jquery-1.4.4.js;Scripts\jquery-
ui.js;Scripts\jquery.unobtrusive-ajax.js;Scripts\jquery.validate-
vsdoc.js;Scripts\jquery.validate.js;Scripts\jquery.validate.unobtrusive.js;Scr
ipts\MicrosoftAjax.debug.js;Scripts\MicrosoftMvcValidation.debug.js">
      <FromTarget>ContactManager.Mvc.wpp.targets</FromTarget>
    </ExcludeFromPackageFiles>
  </ItemGroup>
</Project>
```

9. Save and close the *[project name].wpp.targets* file.

The next time you build and package your web application project, the WPP will automatically detect the *.wpp.targets* file. Any files and folders you specified will not be included in the web package.

Conclusion

This topic described how to exclude specific files and folders when you build a web package, by creating a custom *.wpp.targets* file in the same folder as your web application project file.

Further Reading

For more information on using custom Microsoft Build Engine (MSBuild) project files to control the deployment process, see Understanding the Project File and Understanding the Build Process. For more information on the packaging and deployment process, see Building and Packaging Web Application Projects, Configuring Parameters for Web Package Deployment, and Deploying Web Packages.

Excluding Files and Folders from Deployment

This topic describes how you can exclude files and folders from a web deployment package when you build and package a web application project.

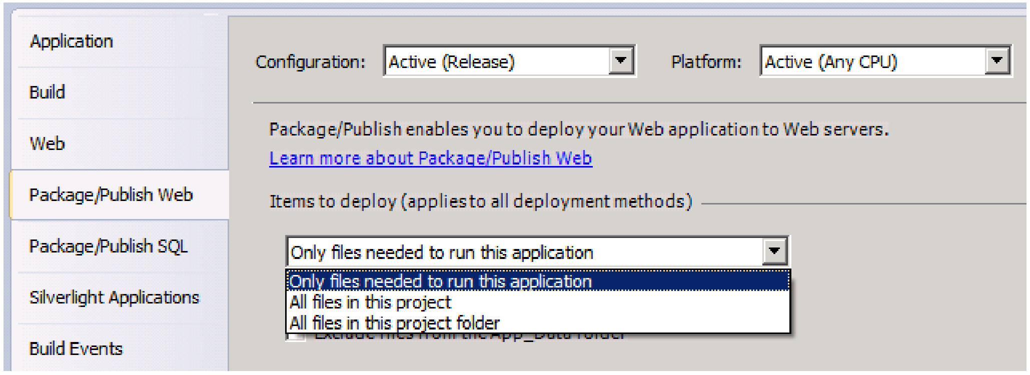
Overview

When you build a web application project in Visual Studio 2010, the Web Publishing Pipeline (WPP) lets you extend this build process by packaging your compiled web application into a deployable web package. You can then use the Internet Information Services (IIS) Web Deployment Tool (Web Deploy) to deploy this web package to a remote IIS web server, or import the web package manually through IIS Manager. This packaging process is explained in Building and Packaging Web Application Projects.

So how do you control what gets included in your web package? The project settings in Visual Studio, through the underlying project file, provide sufficient control for a lot of scenarios. However, in some cases you may want to tailor the contents of your web package to specific destination environments. For example, you might want to include a folder for log files when you deploy your application to a test environment but exclude the folder when you deploy the application to a staging or production environment. This topic will show you how to do this.

What Gets Included by Default?

When you configure your web application project properties in Visual Studio, the **Items to deploy** list on the **Package/Publish Web** page lets you specify what you want to include in your web deployment package. By default, this is set to **Only files needed to run this application**.


Application
Build
Web
Package/Publish Web
Package/Publish SQL
Silverlight Applications
Build Events
Only files needed to run this application
All files in this project
All files in this project folder

Configuration: Active (Release)

Platform: Active (Any CPU)

Package/Publish enables you to deploy your Web application to Web servers.

Learn more about Package/Publish Web

Items to deploy (applies to all deployment methods)

Only files needed to run this application

When you choose **Only files needed to run this application**, the WPP will try to determine which files should be added to the web package. This includes:

- All the build outputs for the project.
- Any files marked with a build action of **Content**.

Note: The logic that determines which files to include is contained in this file:

%PROGRAMFILES%\MSBuild\Microsoft\VisualStudio\v10.0\Web\ Microsoft.Web.Publishing.OnlyFilesToRunTheApp.targets

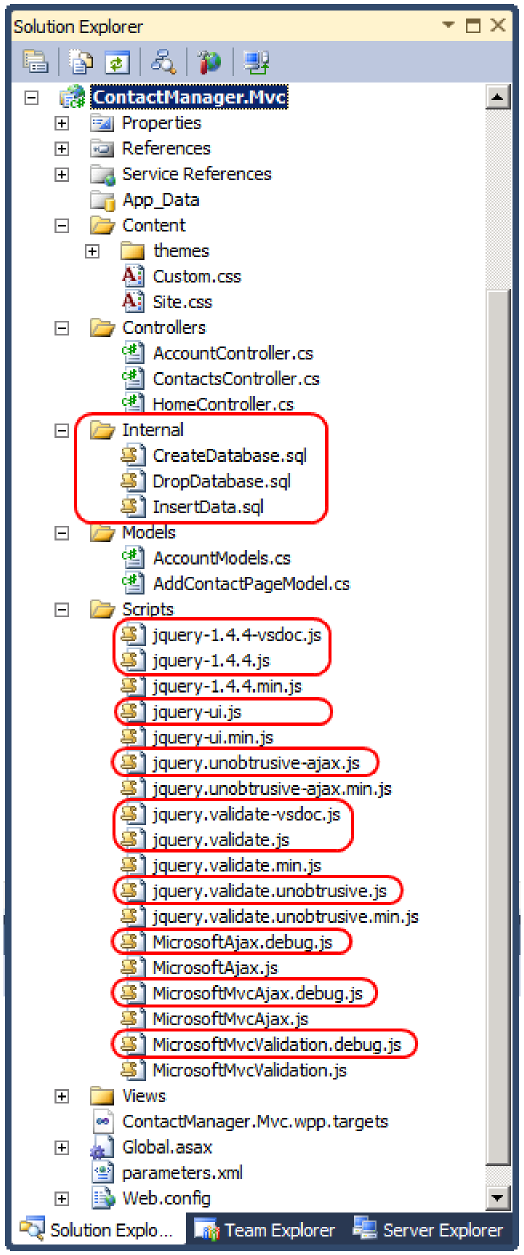
Excluding Specific Files and Folders

In some cases, you'll want more fine-grained control over which files and folders are deployed. If you know which files you want to exclude ahead of time, and the exclusion applies to all destination environments, you can simply set the **Build Action** of each file to **None**.

To exclude specific files from deployment

1. In the **Solution Explorer** window, right-click the file, and then click **Properties**.
2. In the **Properties** window, in the **Build Action** row, select **None**.

However, this approach is not always convenient. For example, you may want to vary which files and folders are included according to your destination environment, and from outside Visual Studio. For example, in the Contact Manager sample solution, take a look at the contents of the ContactManager.Mvc project:



```
Solution Explorer
ContactManager.Mvc
    Properties
    References
    Service References
    App_Data
    Content
        themes
        Custom.css
        Site.css
    Controllers
        AccountController.cs
        ContactsController.cs
        HomeController.cs
    Internal
        CreateDatabase.sql
        DropDatabase.sql
        InsertData.sql
    Models
        AccountModels.cs
        AddContactPageModel.cs
    Scripts
        jquery-1.4.4-vsdoc.js
        jquery-1.4.4.js
        jquery-1.4.4.min.js
        jquery-ui.js
        jquery-ui.min.js
        jquery.unobtrusive-ajax.js
        jquery.unobtrusive-ajax.min.js
        jquery.validate-vsdoc.js
        jquery.validate.js
        jquery.validate.min.js
        jquery.validate.unobtrusive.js
        jquery.validate.unobtrusive.min.js
        MicrosoftAjax.debug.js
        MicrosoftAjax.js
        MicrosoftMvcAjax.debug.js
        MicrosoftMvcAjax.js
        MicrosoftMvcValidation.debug.js
        MicrosoftMvcValidation.js
    Views
    ContactManager.Mvc.wpp.targets
    Global.asax
    parameters.xml
    Web.config
```

- The Internal folder contains some SQL scripts that the developer uses to create, drop, and populate local databases for development purposes. Nothing in this folder should be deployed to a staging or production environment.

- The Scripts folder contains several JavaScript files. A lot of these files are included purely to support debugging or provide IntelliSense in Visual Studio. Some of these files should not be deployed to staging or production environments. However, you may want to deploy them to a developer test environment to facilitate troubleshooting.

Although you could manipulate your project files to exclude specific files and folders, there is an easier way. The WPP includes a mechanism to exclude files and folders by building item lists named **ExcludeFromPackageFolders** and **ExcludeFromPackageFiles**. You can extend this mechanism by adding your own items to these lists. To do this, you need to complete these high-level steps:

1. Create a custom project file named *[project name].wpp.targets* in the same folder as your project file.

   > **Note:** The *.wpp.targets* file needs to go in the same folder as your web application project file—for example, *ContactManager.Mvc.csproj*—rather than in the same folder as any custom project files you use to control the build and deployment process.

2. In the *.wpp.targets* file, add an **ItemGroup** element.
3. In the **ItemGroup** element, add **ExcludeFromPackageFolders** and **ExcludeFromPackageFiles** items to exclude specific files and folders as required.

This is the basic structure of this *.wpp.targets* file:

XML

```xml
<Project ToolsVersion="4.0"
         xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <ItemGroup>
    <ExcludeFromPackageFolders Include="[semi-colon-separated folder list]">
      <FromTarget>[arbitrary metadata value]</FromTarget>
    </ExcludeFromPackageFolders>
    <ExcludeFromPackageFiles Include="[semi-colon-separated file list]">
      <FromTarget>[arbitrary metadata value]</FromTarget>
    </ExcludeFromPackageFiles>
  </ItemGroup>
</Project>
```

Note that each item includes an item metadata element named **FromTarget**. This is an optional value that doesn't affect the build process; it simply serves to indicate why particular files or folders were omitted if someone reviews the build logs.

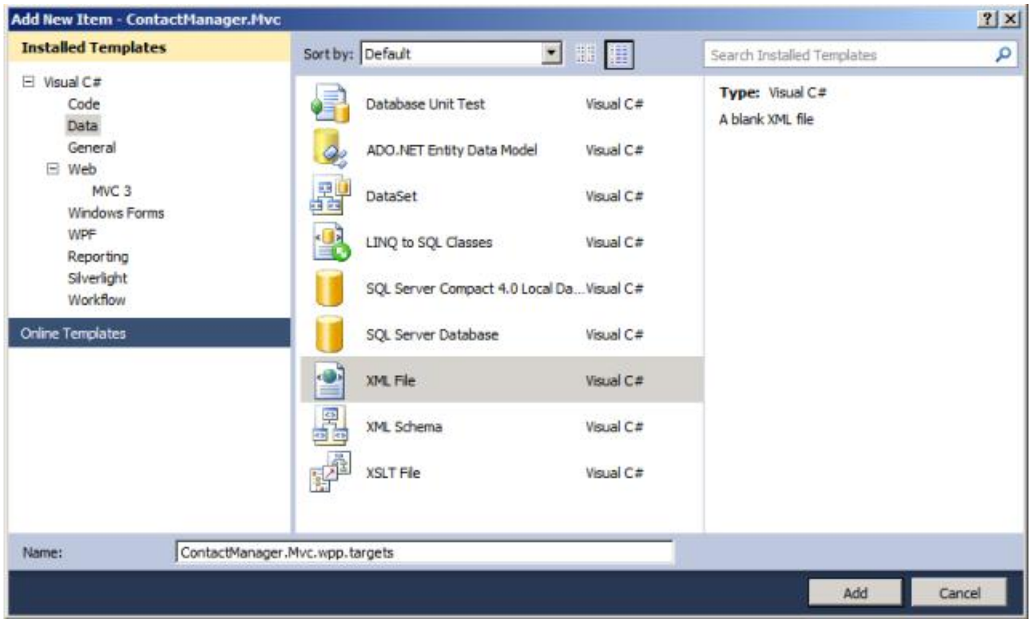
Excluding Files and Folders from a Web Package

The next procedure shows you how to add a *.wpp.targets* file to a web application project and how to use the file to exclude specific files and folders from the web package when you build your project.

To exclude files and folders from a web deployment package

1. Open your solution in Visual Studio 2010.
2. In the **Solution Explorer** window, right-click your web application project node (for example, **ContactManager.Mvc**), point to **Add**, and then click **New Item**.
3. In the **Add New Item** dialog box, select the **XML File** template.

4. In the **Name** box, type *[project name]***.wpp.targets** (for example, **ContactManager.Mvc.wpp.targets**), and then click **Add**.



Add New Item - ContactManager.Mvc

Sort by: Default

Name: ContactManager.Mvc.wpp.targets

Type: Visual C#

A blank XML file

> **Note:** If you add a new item to the root node of a project, the file is created in the same folder as the project file. You can verify this by opening the folder in Windows Explorer.

5. In the file, add a **Project** element and an **ItemGroup** element:

XML

```
<Project ToolsVersion="4.0"
         xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <ItemGroup>
  </ItemGroup>
</Project>
```

6. If you want to exclude folders from the web package, add an **ExcludeFromPackageFolders** element to the **ItemGroup** element:

   a. In the **Include** attribute, provide a semicolon-separated list of the folders you want to exclude.

   b. In the **FromTarget** metadata element, provide a meaningful value to indicate why the folders are being excluded, like the name of the *.wpp.targets* file.

XML

```
<ExcludeFromPackageFolders Include="Internal">
  <FromTarget>ContactManager.Mvc.wpp.targets</FromTarget>
</ExcludeFromPackageFolders>
```

7. If you want to exclude files from the web package, add an **ExcludeFromPackageFiles** element to the **ItemGroup** element:

   a. In the **Include** attribute, provide a semicolon-separated list of the files you want to exclude.

   b. In the **FromTarget** metadata element, provide a meaningful value to indicate why the files are being excluded, like the name of the *.wpp.targets* file.

   XML

   ```xml
   <ExcludeFromPackageFiles Include="Scripts\jquery-1.4.4-
   vsdoc.js;Scripts\jquery-1.4.4.js;Scripts\jquery-
   ui.js;Scripts\jquery.unobtrusive-ajax.js;Scripts\jquery.validate-
   vsdoc.js;Scripts\jquery.validate.js;Scripts\jquery.validate.unobtrusive.js;Scr
   ipts\MicrosoftAjax.debug.js;Scripts\MicrosoftMvcValidation.debug.js">
     <FromTarget>ContactManager.Mvc.wpp.targets</FromTarget>
   </ExcludeFromPackageFiles>
   ```

8. The *[project name].wpp.targets* file should now resemble this:

   XML

   ```xml
   <Project ToolsVersion="4.0"
            xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
     <ItemGroup>
       <ExcludeFromPackageFolders Include="Internal">
         <FromTarget>ContactManager.Mvc.wpp.targets</FromTarget>
       </ExcludeFromPackageFolders>
       <ExcludeFromPackageFiles Include="Scripts\jquery-1.4.4-
   vsdoc.js;Scripts\jquery-1.4.4.js;Scripts\jquery-
   ui.js;Scripts\jquery.unobtrusive-ajax.js;Scripts\jquery.validate-
   vsdoc.js;Scripts\jquery.validate.js;Scripts\jquery.validate.unobtrusive.js;Scr
   ipts\MicrosoftAjax.debug.js;Scripts\MicrosoftMvcValidation.debug.js">
         <FromTarget>ContactManager.Mvc.wpp.targets</FromTarget>
       </ExcludeFromPackageFiles>
     </ItemGroup>
   </Project>
   ```

9. Save and close the *[project name].wpp.targets* file.

The next time you build and package your web application project, the WPP will automatically detect the *.wpp.targets* file. Any files and folders you specified will not be included in the web package.

Conclusion

This topic described how to exclude specific files and folders when you build a web package, by creating a custom *.wpp.targets* file in the same folder as your web application project file.

Further Reading

For more information on using custom Microsoft Build Engine (MSBuild) project files to control the deployment process, see Understanding the Project File and Understanding the Build Process. For more information on the packaging and deployment process, see Building and Packaging Web Application Projects, Configuring Parameters for Web Package Deployment, and Deploying Web Packages.

Taking Web Applications Offline with Web Deploy

This topic describes how to take a web application offline for the duration of an automated deployment using the Internet Information Services (IIS) Web Deployment Tool (Web Deploy). Users who browse to the web application are redirected to an *App_offline.htm* file until the deployment is complete.

Task Overview

In a lot of scenarios, you'll want to take a web application offline while you make changes to related components, like databases or web services. Typically, in IIS and ASP.NET, you accomplish this by placing a file named *App_offline.htm* in the root folder of the IIS website or web application. The *App_offline.htm* file is a standard HTML file and will usually contain a simple message advising the user that the site is temporarily unavailable due to maintenance. While the *App_offline.htm* file exists in the root folder of the website, IIS will automatically redirect any requests to the file. When you've finished making updates, you remove the *App_offline.htm* file and the website resumes serving requests as usual.

When you use Web Deploy to perform automated or single-step deployments to a target environment, you may want to incorporate adding and removing the *App_offline.htm* file into your deployment process. To do this, you'll need to complete these high-level tasks:

- In the Microsoft Build Engine (MSBuild) project file that you use to control the deployment process, create an MSBuild target that copies an *App_offline.htm* file to the destination server before any deployment tasks begin.
- Add another MSBuild target that removes the *App_offline.htm* file from the destination server when all deployment tasks are complete.
- In the web application project, create a *.wpp.targets* file that ensures that an *App_offline.htm* file is added to the deployment package when Web Deploy is invoked.

This topic will show you how to perform these procedures. The tasks and walkthroughs in this topic assume that you've already created a solution that contains at least one web application project, and that you use a custom project file to control the deployment process as described in Web Deployment in the Enterprise. Alternatively, you can use the Contact Manager sample solution to follow the examples in the topic.

Adding an App_Offline File to a Web Application Project

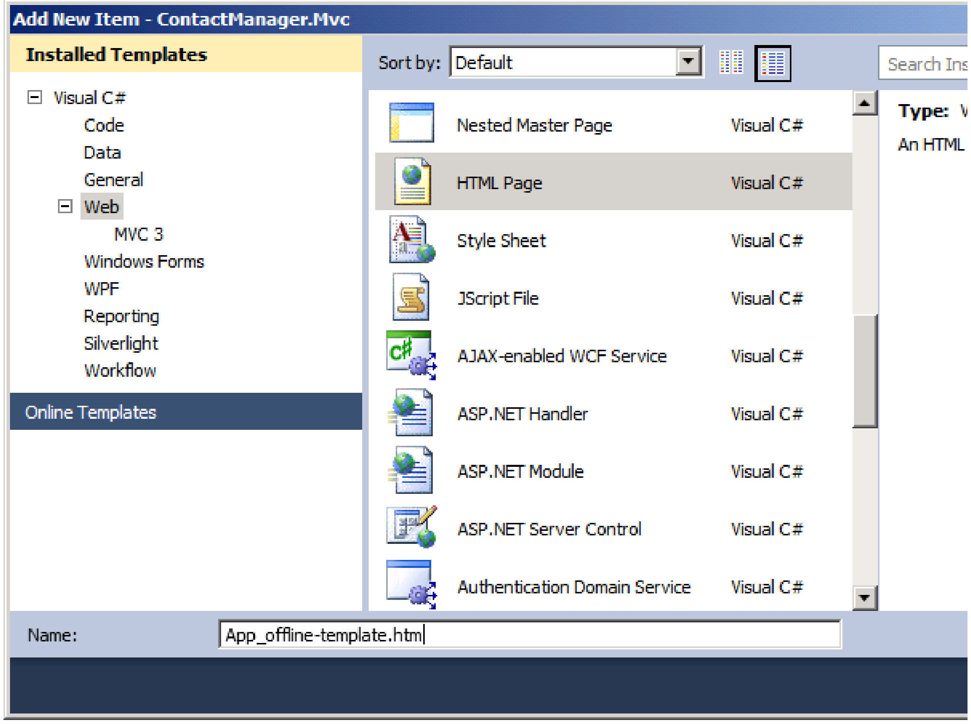
The first task you need to complete is to add an *App_offline* file to your web application project:

- To prevent the file from interfering with the development process (you don't want your application to be permanently offline), you should call it something other than *App_offline.htm*. For example, you could name the file *App_offline-template.htm*.

- To prevent the file from being deployed as-is, you should set the build action to **None**.

To add an App_offline file to a web application project

1. Open your solution in Visual Studio 2010.
2. In the **Solution Explorer** window, right-click your web application project, point to **Add**, and then click **New Item**.
3. In the **Add New Item** dialog box, select **HTML Page**.
4. In the **Name** box, type **App_offline-template.htm**, and then click **Add**.



Sort by: Default

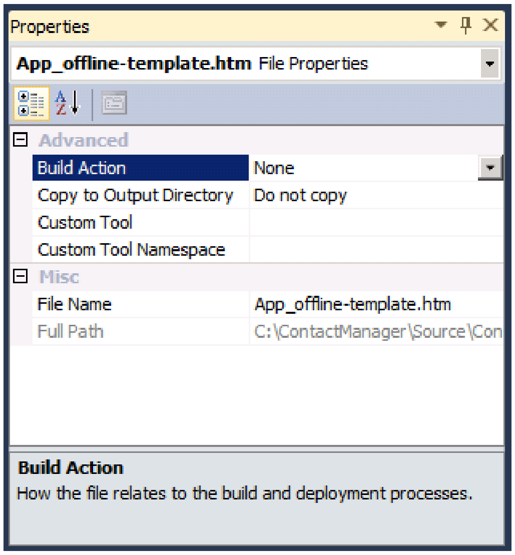
Nested Master Page	Visual C#
HTML Page	Visual C#
Style Sheet	Visual C#
JScript File	Visual C#
AJAX-enabled WCF Service	Visual C#
ASP.NET Handler	Visual C#
ASP.NET Module	Visual C#
ASP.NET Server Control	Visual C#
Authentication Domain Service	Visual C#

Name: App_offline-template.htm

5. Add some simple HTML to inform users that the application is unavailable, and then save the file. Do not include any server-side tags (for example, any tags that are prefixed with "asp:").

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Application Offline</title>
</head>
<body>
    <p>This web application is temporarily offline for maintenance.</p>
</body>
</html>
```

6. In the **Solution Explorer** window, right-click the new file, and then click **Properties**.
7. In the **Properties** window, in the **Build Action** row, select **None**.



Properties

App_offline-template.htm File Properties

Advanced

Build Action	None
Copy to Output Directory	Do not copy
Custom Tool	
Custom Tool Namespace	

Misc

File Name	App_offline-template.htm
Full Path	C:\ContactManager\Source\Con

Build Action
How the file relates to the build and deployment processes.

Deploying and Deleting an App_Offline File

The next step is to modify your deployment logic to copy the file to the destination server at the start of the deployment process and remove it at the end.

> **Note:** The next procedure assumes that you're using a custom MSBuild project file to control your deployment process, as described in Understanding the Project File. If you're deploying direct from Visual Studio, you'll need to use a different approach. Sayed Ibrahim Hashimi describes one such approach in How to Take Your Web App Offline During Publishing.

To deploy an *App_offline* file to a destination IIS website, you need to invoke MSDeploy.exe using the Web Deploy **contentPath** provider. The **contentPath** provider supports both physical directory paths and IIS website or application paths, which makes it the ideal choice for synchronizing a file between a Visual Studio project folder and an IIS web application. To deploy the file, your MSDeploy command should resemble this:

```
msdeploy.exe –verb:sync
             -source:contentPath="[Project folder]\App_offline.template.htm"
             -dest:contentPath="[IIS application path]/App_offline.htm",
              computerName="[Destination web server]"
```

To remove the file from the destination site at the end of the deployment process, your MSDeploy command should resemble this:

```
msdeploy.exe –verb:delete
             -dest:contentPath="[IIS application path]/App_offline.htm",
              computerName="[Destination web server]"
```

To automate these commands as part of a build and deployment process, you need to integrate them into your custom MSBuild project file. The next procedure describes how to do this.

To deploy and delete an App_offline file

1. In Visual Studio 2010, open the MSBuild project file that controls your deployment process. In the Contact Manager sample solution, this is the *Publish.proj* file.

2. In the root **Project** element, create a new **PropertyGroup** element to store variables for the *App_offline* deployment:

   XML

   ```
   <PropertyGroup>
     <AppOfflineTemplateFilename
       Condition=" '$(AppOfflineTemplateFilename)'=='' ">
         app_offline-template.htm
     </AppOfflineTemplateFilename>
     <AppOfflineSourcePath
       Condition=" '$(AppOfflineSourcePath)'==''">
         $(SourceRoot)ContactManager.Mvc\$(AppOfflineTemplateFilename)
     </AppOfflineSourcePath>
   </PropertyGroup>
   ```

   The **SourceRoot** property is defined elsewhere in the *Publish.proj* file. It indicates the location of the root folder for the source content relative to the current path—in other words, relative to the location of the *Publish.proj* file.

   The **contentPath** provider will not accept relative file paths, so you need to get an absolute path to your source file before you can deploy it. You can use the ConvertToAbsolutePath task to do this.

3. Add a new **Target** element named **GetAppOfflineAbsolutePath**. Within this target, use the **ConvertToAbsolutePath** task to get an absolute path to the *App_offline-template* file in your project folder.

   XML

   ```
   <Target Name="GetAppOfflineAbsolutePath" BeforeTargets="DeployAppOffline">
     <ConvertToAbsolutePath Paths="$(AppOfflineSourcePath)">
       <Output TaskParameter="AbsolutePaths"
               PropertyName="AppOfflineAbsoluteSourcePath" />
     </ConvertToAbsolutePath>
   </Target>
   ```

   This target takes the relative path to the *App_offline-template* file in your project folder and saves it to a new property as an absolute file path. The **BeforeTargets** attribute specifies that you want this target to execute before the **DeployAppOffline** target, which you'll create in the next step.

4. Add a new target named **DeployAppOffline**. Within this target, invoke the MSDeploy.exe command that deploys your *App_offline* file to the destination web server.

   XML

```
<Target Name="DeployAppOffline"
        Condition=" '$(EnableAppOffline'!='false' ">
  <PropertyGroup>
    <_Cmd>"$(MSDeployPath)\msdeploy.exe" -verb:sync
           -source:contentPath="$(AppOfflineAbsoluteSourcePath)"
           -dest:contentPath="$(ContactManagerIisPath)/App_offline.htm",
            computerName="$(MSDeployComputerName)"
    </_Cmd>
  </PropertyGroup>
  <Exec Command="$(_Cmd)"/>
</Target>
```

In this example, the **ContactManagerIisPath** property is defined elsewhere in the project file. This is simply an IIS application path, in the form *[IIS Website Name]/[Application Name]*. Including a condition in the target enables users to switch the *App_offline* deployment on or off by changing a property value or providing a command-line parameter.

5. Add a new target named **DeleteAppOffline**. Within this target, invoke the MSDeploy.exe command that removes your *App_offline* file from the destination web server.

XML

```
<Target Name="DeleteAppOffline"
        Condition=" '$(EnableAppOffline'!='false' ">
  <PropertyGroup>
    <_Cmd>"$(MSDeployPath)\msdeploy.exe" -verb:delete
           -dest:contentPath="$(ContactManagerIisPath)/App_offline.htm",
            computerName="$(MSDeployComputerName)"
    </_Cmd>
  </PropertyGroup>
  <Exec Command="$(_Cmd)"/>
</Target>
```

The final task is to invoke these new targets at appropriate points during the execution of your project file. You can do this in various ways. For example, in the *Publish.proj* file, the **FullPublishDependsOn** property specifies a list of targets that must be executed in order when the **FullPublish** default target is invoked.

6. Modify your MSBuild project file to invoke the **DeployAppOffline** and **DeleteAppOffline** targets at appropriate points in the publishing process.

XML

```
<PropertyGroup>
  <FullPublishDependsOn>
    Clean;
    BuildProjects;
    DeployAppOffline;
    GatherPackagesForPublishing;
    PublishDbPackages;
    DeployTestDBPermissions;
    PublishWebPackages;
    DeleteAppOffline;
  </FullPublishDependsOn>
</PropertyGroup>
<Target Name="FullPublish" DependsOnTargets="$(FullPublishDependsOn)" />
```

When you run your custom MSBuild project file, the *App_offline* file will be deployed to the server immediately after a successful build. It will then be deleted from the server once all the deployment tasks are complete.

Adding an App_Offline File to Deployment Packages

Depending on how you configure your deployment, any existing content at the destination IIS web application—like the *App_offline.htm* file—may be deleted automatically when you deploy a web package to the destination. To ensure that the *App_offline.htm* file remains in place for the duration of the deployment, you need to include the file within the web deployment package itself in addition to deploying the file directly at the start of the deployment process.

If you've followed the previous tasks in this topic, you'll have added the *App_offline.htm* file to your web application project under a different filename (we used *App_offline-template.htm*) and you'll have set the build action to **None**. These changes are necessary to prevent the file from interfering with development and debugging. As a result, you need to customize the packaging process to ensure that the *App_offline.htm* file is included in the web deployment package.

The Web Publishing Pipeline (WPP) uses an item list named **FilesForPackagingFromProject** to build a list of files that should be included in the web deployment package. You can customize the contents of your web packages by adding your own items to this list. To do this, you need to complete these high-level steps:

1. Create a custom project file named *[project name].wpp.targets* in the same folder as your project file.

   > **Note:** The *.wpp.targets* file needs to go in the same folder as your web application project file—for example, *ContactManager.Mvc.csproj*—rather than in the same folder as any custom project files you use to control the build and deployment process.

2. In the *.wpp.targets* file, create a new MSBuild target that executes *before* the **CopyAllFilesToSingleFolderForPackage** target. This is the WPP target that builds a list of things to include in the package.
3. In the new target, create an **ItemGroup** element.
4. In the **ItemGroup** element, add a **FilesForPackagingFromProject** item and specify the *App_offline.htm* file.

The *.wpp.targets* file should resemble this:

XML

```
<Project ToolsVersion="4.0"
         xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <Target Name="AddAppOfflineToPackage"
          BeforeTargets="CopyAllFilesToSingleFolderForPackage">
    <ItemGroup>
      <FilesForPackagingFromProject Include="App_offline-template.htm">
        <DestinationRelativePath>App_offline.htm</DestinationRelativePath>
    </FilesForPackagingFromProject>
  </ItemGroup>
  </Target>
</Project>
```

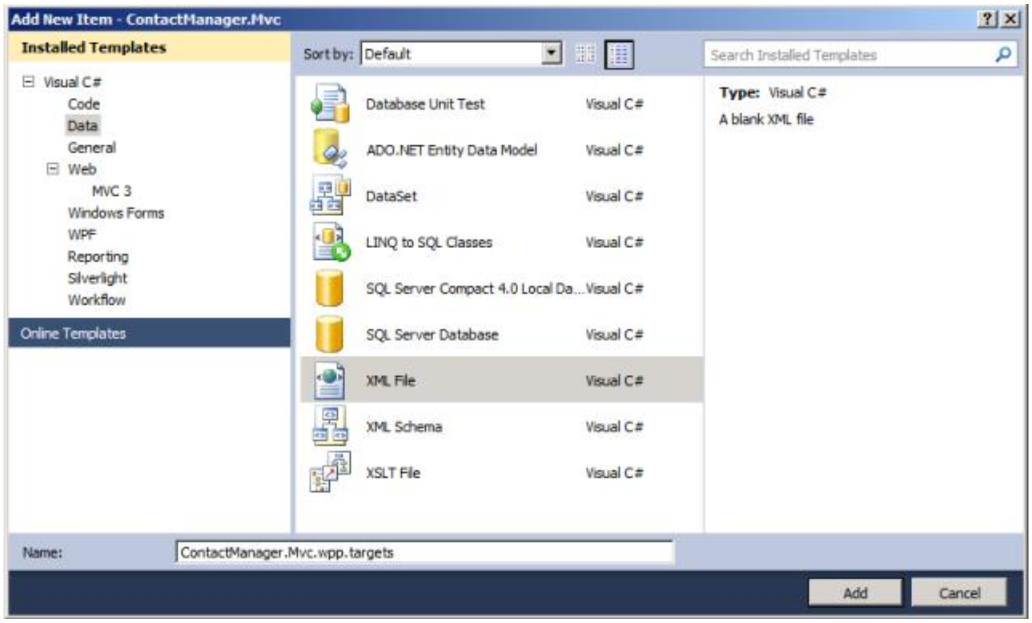
These are the key points of note in this example:

- The **BeforeTargets** attribute inserts this target into the WPP by specifying that it should be executed immediately before the **CopyAllFilesToSingleFolderForPackage** target.
- The **FilesForPackagingFromProject** item uses the **DestinationRelativePath** metadata value to rename the file from *App_offline-template.htm* to *App_offline.htm* as it's added to the list.

The next procedure shows you how to add this *.wpp.targets* file to a web application project.

To add a .wpp.targets file to a web deployment package

1. Open your solution in Visual Studio 2010.
2. In the **Solution Explorer** window, right-click your web application project node (for example, **ContactManager.Mvc**), point to **Add**, and then click **New Item**.
3. In the **Add New Item** dialog box, select the **XML File** template.
4. In the **Name** box, type *[project name]***.wpp.targets** (for example, **ContactManager.Mvc.wpp.targets**), and then click **Add**.



Name: ContactManager.Mvc.wpp.targets

> **Note:** If you add a new item to the root node of a project, the file is created in the same folder as the project file. You can verify this by opening the folder in Windows Explorer.

5. In the file, add the MSBuild markup described previously.

XML

```
<Project ToolsVersion="4.0"
         xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <Target Name="AddAppOfflineToPackage"
          BeforeTargets="CopyAllFilesToSingleFolderForPackage">
    <ItemGroup>
      <FilesForPackagingFromProject Include="App_offline-template.htm">
        <DestinationRelativePath>App_offline.htm</DestinationRelativePath>
    </FilesForPackagingFromProject>
  </ItemGroup>
  </Target>
</Project>
```

6. Save and close the *[project name].wpp.targets* file.

The next time you build and package your web application project, the WPP will automatically detect the *.wpp.targets* file. The *App_offline-template.htm* file will be included in the resulting web deployment package as *App_offline.htm*.

> **Note:** If your deployment fails, the *App_offline.htm* file will remain in place and your application will remain offline. This is typically the desired behavior. To bring your application back online, you can delete the *App_offline.htm* file from your web server. Alternatively, if you correct any errors and run a successful deployment, the *App_offline.htm* file will be removed.

Conclusion

This topic described how to take a web application offline for the duration of a deployment, by publishing an *App_offline.htm* file to the destination server at the start of the deployment process and removing it at the end. It also covered how to include an *App_offline.htm* file in a web deployment package.

Further Reading

For more information on the packaging and deployment process, see Building and Packaging Web Application Projects, Configuring Parameters for Web Package Deployment, and Deploying Web Packages.

If you publish your web applications directly from Visual Studio, rather than using the custom MSBuild project file approach described in these tutorials, you'll need to use a slightly different approach to take your application offline during the publishing process. For more information, see How to take your web app offline during publishing (blog post).

Running Windows PowerShell Scripts from MSBuild Project Files

This topic describes how to run a Windows PowerShell script as part of a build and deployment process. You can run a script locally (in other words, on the build server) or remotely, like on a destination web server or database server.

There are lots of reasons why you might want to run a post-deployment Windows PowerShell script. For example, you might want to:

- Add a custom event source to the registry.
- Generate a file system directory for uploads.
- Clean up build directories.
- Write entries to a custom log file.
- Send emails inviting users to a newly provisioned web application.
- Create user accounts with the appropriate permissions.
- Configure replication between SQL Server instances.

This topic will show you how to run Windows PowerShell scripts both locally and remotely from a custom target in a Microsoft Build Engine (MSBuild) project file.

Task Overview

To run a Windows PowerShell script as part of an automated or single-step deployment process, you'll need to complete these high-level tasks:

- Add the Windows PowerShell script to your solution and to source control.

- Create a command that invokes your Windows PowerShell script.
- Escape any reserved XML characters in your command.
- Create a target in your custom MSBuild project file and use the **Exec** task to run your command.

This topic will show you how to perform these procedures. The tasks and walkthroughs in this topic assume that you're already familiar with MSBuild targets and properties, and that you understand how to use a custom MSBuild project file to drive a build and deployment process. For more information, see Understanding the Project File and Understanding the Build Process.

Creating and Adding Windows PowerShell Scripts

The tasks in this topic use a sample Windows PowerShell script named **LogDeploy.ps1** to illustrate how to run scripts from MSBuild. The **LogDeploy.ps1** script contains a simple function that writes a single-line entry to a log file:

Windows PowerShell

```
function LogDeployment
{
  param([string]$filepath,[string]$deployDestination)
  $datetime = Get-Date
  $filetext = "Deployed package to " + $deployDestination + " on " + $datetime
  $filetext | Out-File -filepath $filepath -Append
}

LogDeployment $args[0] $args[1]
```

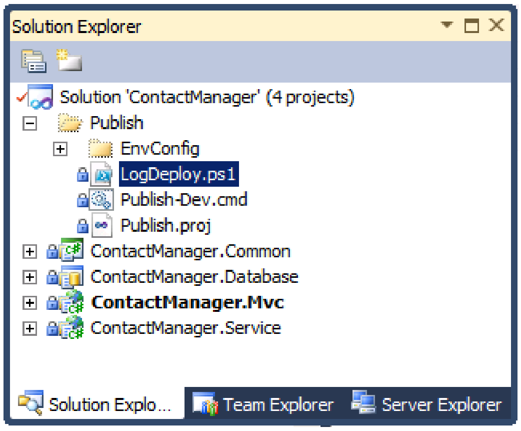
The **LogDeploy.ps1** script accepts two parameters. The first parameter represents the full path to the log file to which you want to add an entry, and the second parameter represents the deployment destination that you want to record in the log file. When you run the script, it adds a line to the log file in this format:

```
Deployed package to TESTWEB1 on 02/11/2012 09:28:18
```

To make the **LogDeploy.ps1** script available to MSBuild, you need to:

- Add the script to source control.
- Add the script to your solution in Visual Studio 2010.

You don't need to deploy the script with your solution content, regardless of whether you plan to run the script on the build server or on a remote computer. One option is to add the script to a solution folder. In the Contact Manager example, because you want to use the Windows PowerShell script as part of the deployment process, it makes sense to add the script to the Publish solution folder.



The contents of solution folders are copied to build servers as source material. However, they form no part of any project output.

Executing a Windows PowerShell Script on the Build Server

In some scenarios, you may want to run Windows PowerShell scripts on the computer that builds your projects. For example, you might use a Windows PowerShell script to clean up build folders or write entries to a custom log file.

In terms of syntax, running a Windows PowerShell script from an MSBuild project file is the same as running a Windows PowerShell script from a regular command prompt. You need to invoke the powershell.exe executable and use the **–command** switch to provide the commands you want Windows PowerShell to run. (In Windows PowerShell v2, you can also use the **–file** switch). The command should take this format:

```
powershell.exe –command "& { [Path to script] 'parameter1' 'parameter2' ... }"
```

For example:

```
powershell.exe –command
  "& { C:\LogDeploy.ps1 'C:\DeployLogs\log.txt' 'TESTWEB1' }"
```

If the path to your script includes spaces, you need to enclose the file path in single quotes preceded by an ampersand. You can't use double quotes, because you've already used them to enclose the command:

```
powershell.exe –command
  "& { &'C:\Path With Spaces\LogDeploy.ps1'
        'C:\Path With Spaces\log.txt'
        'TESTWEB1' }"
```

There are a few additional considerations when you invoke this command from MSBuild. First, you should include the **–NonInteractive** flag to ensure that the script executes quietly. Next, you should include the **–ExecutionPolicy** flag with an appropriate argument value. This specifies the execution policy that Windows PowerShell will apply to your script and allows you to override the default execution policy, which may prevent execution of your script. You can choose from these argument values:

- A value of **Unrestricted** will allow Windows PowerShell to execute your script, regardless of whether the script is signed.
- A value of **RemoteSigned** will allow Windows PowerShell to execute unsigned scripts that were created on the local machine. However, scripts that were created elsewhere must be signed. (In practice, you're very unlikely to have created a Windows PowerShell script locally on a build server).
- A value of **AllSigned** will allow Windows PowerShell to execute signed scripts only.

The default execution policy is **Restricted**, which prevents Windows PowerShell from running any script files.

Finally, you need to escape any reserved XML characters that occur in your Windows PowerShell command:

- Replace single quotes with **&apos;**
- Replace double quotes with **&quot;**
- Replace ampersands with **&amp;**

When you make these changes, your command will resemble this:

```
powershell.exe -NonInteractive -ExecutionPolicy Unrestricted
               -command &quot;&amp; { &amp;&apos;[Path to script]&apos;
                        &apos;[parameter1]&apos;
                        &apos;[parameter2]&apos; } &quot;
```

Within your custom MSBuild project file, you can create a new target and use the **Exec** task to run this command:

XML

```xml
<Target Name="WriteLogEntry" Condition=" '$(WriteLogEntry)'!='false' ">
  <PropertyGroup>
    <PowerShellExe Condition=" '$(PowerShellExe)'=='' ">
      %WINDIR%\System32\WindowsPowerShell\v1.0\powershell.exe
    </PowerShellExe>
    <ScriptLocation Condition=" '$(ScriptLocation)'=='' ">
      C:\Path With Spaces\LogDeploy.ps1
    </ScriptLocation>
    <LogFileLocation Condition=" '$(LogFileLocation)'=='' ">
      C:\Path With Spaces\ContactManagerDeployLog.txt
    </LogFileLocation>
  </PropertyGroup>
  <Exec Command="$(PowerShellExe) -NonInteractive -executionpolicy Unrestricted
```

```
                 -command &quot;&amp; {
                          &amp;&apos;$(ScriptLocation)&apos;
                          &apos;$(LogFileLocation)&apos;
                          &apos;$(MSDeployComputerName)&apos;} &quot;" />
</Target>
```

In this example, note that:

- Any variables, like parameter values and the location of the Windows PowerShell executable, are declared as MSBuild properties.
- Conditions are included to enable users to override these values from the command line.
- The **MSDeployComputerName** property is declared elsewhere in the project file.

When you execute this target as part of your build process, Windows PowerShell will run your command and write a log entry to the file you specified.

Executing a Windows PowerShell Script on a Remote Computer

Windows PowerShell is capable of running scripts on remote computers through Windows Remote Management (WinRM). To do this, you need to use the Invoke-Command cmdlet. This lets you execute your script against one or more remote computers without copying the script to the remote computers. Any results are returned to the local computer from which you ran the script.

Note: Before you use the **Invoke-Command** cmdlet to execute Windows PowerShell scripts on a remote computer, you need to configure a WinRM listener to accept remote messages. You can do this by running the command **winrm quickconfig** on the remote computer. For more information, see Installation and Configuration for Windows Remote Management.

From a Windows PowerShell window, you'd use this syntax to run the **LogDeploy.ps1** script on a remote computer:

Windows PowerShell

```
Invoke-Command –ComputerName 'REMOTESERVER1'
               –ScriptBlock { &"C:\Path With Spaces\LogDeploy.ps1"
                               'C:\Path With Spaces\Log.txt'
                               'TESTWEB1' }
```

Note: There are various other ways of using **Invoke-Command** to run a script file, but this approach is the most straightforward when you need to provide parameter values and manage paths with spaces.

When you run this from a command prompt, you need to invoke the Windows PowerShell executable and use the **–command** parameter to provide your instructions:

```
powershell.exe –command
  "& {Invoke-Command –ComputerName 'REMOTESERVER1'
                     –ScriptBlock { &'C:\Path With Spaces\LogDeploy.ps1'
                                     'C:\Path With Spaces\Log.txt'
                                     'TESTWEB1' } "
```

As before, you need to provide some additional switches and escape any reserved XML characters when you run the command from MSBuild:

```
powershell.exe -NonInteractive -executionpolicy Unrestricted
               -command &quot;&amp; Invoke-Command
                 –ComputerName &apos;REMOTESERVER1&apos;
                 -ScriptBlock { &amp;&apos;C:\Path With Spaces\LogDeploy.ps1&apos;
                                &apos; C:\Path With Spaces\Log.txt &apos;
                                &apos;TESTWEB1&apos; } &quot;
```

Finally, as before, you can use the **Exec** task within a custom MSBuild target to execute your command:

XML

```
<Target Name="WriteLogEntry" Condition=" '$(WriteLogEntry)'!='false' ">
  <PropertyGroup>
    <PowerShellExe Condition=" '$(PowerShellExe)'=='' ">
      %WINDIR%\System32\WindowsPowerShell\v1.0\powershell.exe
    </PowerShellExe>
    <ScriptLocation Condition=" '$(ScriptLocation)'=='' ">
      C:\Path With Spaces\LogDeploy.ps1
    </ScriptLocation>
    <LogFileLocation Condition=" '$(LogFileLocation)'=='' ">
      C:\Path With Spaces\ContactManagerDeployLog.txt
    </LogFileLocation>
  </PropertyGroup>
  <Exec Command="$(PowerShellExe) -NonInteractive -executionpolicy Unrestricted
                 -command &quot;&amp; invoke-command -scriptblock {
                          &amp;&apos;$(ScriptLocation)&apos;
                          &apos;$(LogFileLocation)&apos;
                          &apos;$(MSDeployComputerName)&apos;}
                          &quot;"/>
</Target>
```

When you execute this target as part of your build process, Windows PowerShell will run your script on the computer you specified in the **–computername** argument.

Conclusion

This topic described how to run a Windows PowerShell script from an MSBuild project file. You can use this approach to run a Windows PowerShell script, either locally or on a remote computer, as part of an automated or single-step build and deployment process.

Further Reading

For guidance on signing Windows PowerShell scripts and managing execution policies, see Running Windows PowerShell Scripts. For guidance on running Windows PowerShell commands from a remote computer, see Running Remote Commands.

For more information on using custom MSBuild project files to control the deployment process, see Understanding the Project File and Understanding the Build Process.

Troubleshooting the Packaging Process

This topic describes how you can collect detailed information about the packaging process by using the **EnablePackageProcessLoggingAndAssert** property in the Microsoft Build Engine (MSBuild).

When you set the **EnablePackageProcessLoggingAndAssert** property to **true**, MSBuild will:

- Add additional information about the packaging process to the build logs.
- Log errors under certain conditions, for example, if duplicate files are found in the packaging list.
- Create a Log directory in the *ProjectName*_Package folder and use it to record information about the files you're packaging.

If the packaging process is failing, or your web deployment packages don't contain the files that you expect, you can use this information to troubleshoot the process and pinpoint where things are going wrong.

Note: The **EnablePackageProcessLoggingAndAssert** property only works if you build your project using the **Debug** configuration. The property is ignored in other configurations.

Understanding the EnablePackageProcessLoggingAndAssert Property

Building and Packaging Web Application Projects described how the Web Publishing Pipeline (WPP) provides a set of MSBuild targets that extend the functionality of MSBuild and enable it to integrate with the Internet Information Services (IIS) Web Deployment Tool (Web Deploy). When you package a web application project, you're invoking WPP targets.

Lots of these WPP targets include conditional logic that logs additional information when the **EnablePackageProcessLoggingAndAssert** property is set to **true**. For example, if you review the **Package** target, you can see that it creates an additional log directory and writes a list of files to a text file if **EnablePackageProcessLoggingAndAssert** is equal to **true**.

XML

```xml
<Target Name="Package"
        Condition="$(_CreatePackage)"
        DependsOnTargets="$(PackageDependsOn)">

    <!--Log the information  Set $(EnablePackageProcessLoggingAndAssert) to True
        if you want to see this information-->
    <MakeDir Condition="$(EnablePackageProcessLoggingAndAssert) And
                        !Exists('$(PackageLogDir)')"
             Directories="$(PackageLogDir)" />
    <WriteLinesToFile Condition="$(EnablePackageProcessLoggingAndAssert)"
```

```
                      Encoding="utf-8"
                      File="$(PackageLogDir)\Prepackage.txt"
                      Lines="@(FilesForPackagingFromProject->'
                             From:%(Identity)
                             DestinationRelativePath:%(DestinationRelativePath)
                             Exclude:%(Exclude)
                             FromTarget:%(FromTarget)
                             Category:%(Category)
                             ProjectFileType:%(ProjectFileType)')"
                      Overwrite="True" />
```

> **Note:** The WPP targets are defined in the *Microsoft.Web.Publishing.targets* file in the %PROGRAMFILES(x86)%\MSBuild\Microsoft\VisualStudio\v10.0\Web folder. You can open this file and review the targets in Visual Studio 2010 or any XML editor. Take care not to modify the contents of the file.

Enabling the Additional Logging

You can supply a value for the **EnablePackageProcessLoggingAndAssert** property in various ways, depending on how you build your project.

If you build your project from the command line, you can supply a value for the **EnablePackageProcessLoggingAndAssert** property as a command-line argument:

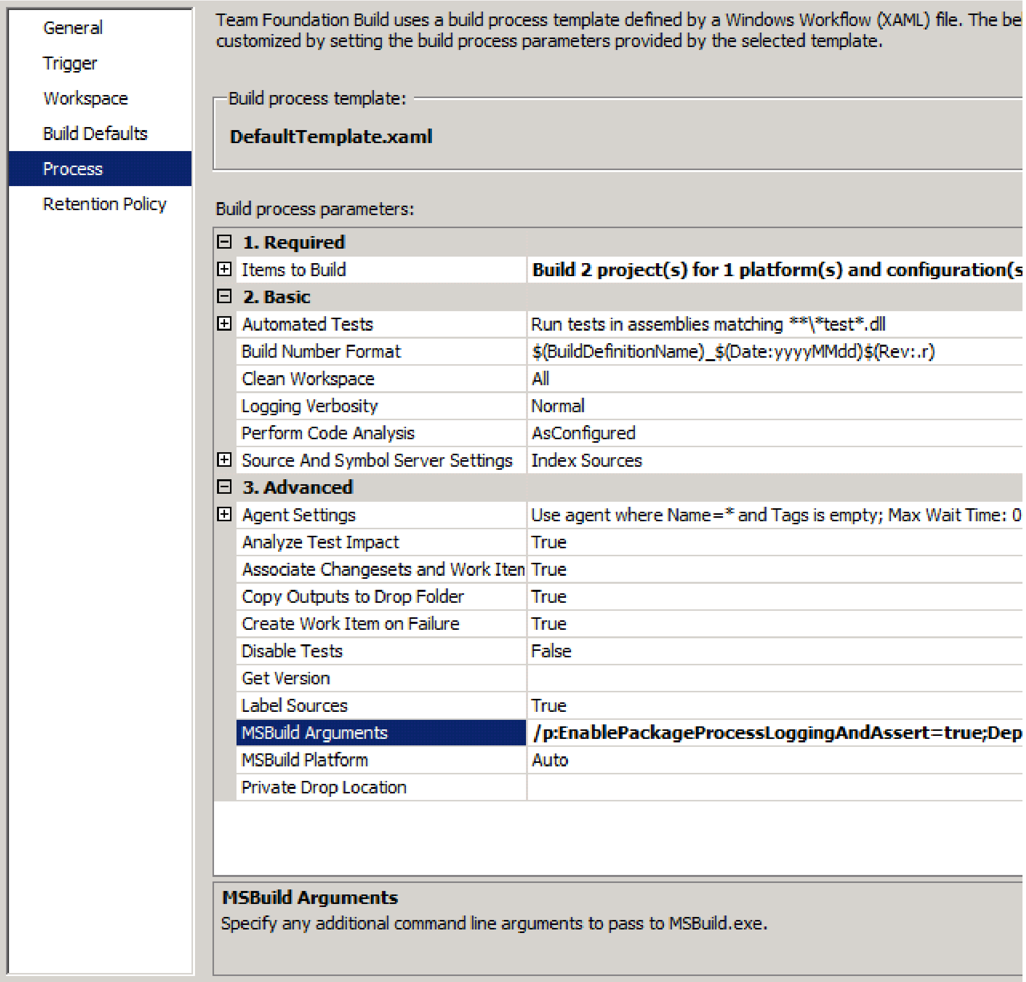
```
MSBuild.exe /t:Build
            /p:Configuration=DEBUG
            /p:DeployOnBuild=true
            /p:DeployTarget=Package
            /p:EnablePackageProcessLoggingAndAssert=true
            [Your project].csproj
```

If you're using a custom project file to build your projects, you can include the **EnablePackageProcessLoggingAndAssert** value in the **Properties** attribute of the **MSBuild** task:

XML

```
<Target Name="BuildProjects" Condition=" '$(BuildingInTeamBuild)'!='true' ">
   <MSBuild Projects="@(ProjectsToBuild)"
            Properties="OutDir=$(OutputRoot);
                        Configuration=$(Configuration);
                        DeployOnBuild=true;
                        DeployTarget=Package;
                        EnablePackageProcessLoggingAndAssert=true"
            Targets="Build" />
  </Target>
```

If you're using a Team Foundation Server (TFS) build definition to build your projects, you can supply a value for the **EnablePackageProcessLoggingAndAssert** property in the **MSBuild Arguments** row:



Team Foundation Build uses a build process template defined by a Windows Workflow (XAML) file. The be
customized by setting the build process parameters provided by the selected template.

Build process template:

DefaultTemplate.xaml

Build process parameters:

⊟ **1. Required**	
⊞ Items to Build	**Build 2 project(s) for 1 platform(s) and configuration(s**
⊟ **2. Basic**	
⊞ Automated Tests	Run tests in assemblies matching ***test*.dll
Build Number Format	$(BuildDefinitionName)_$(Date:yyyyMMdd)$(Rev:.r)
Clean Workspace	All
Logging Verbosity	Normal
Perform Code Analysis	AsConfigured
⊞ Source And Symbol Server Settings	Index Sources
⊟ **3. Advanced**	
⊞ Agent Settings	Use agent where Name=* and Tags is empty; Max Wait Time: 0
Analyze Test Impact	True
Associate Changesets and Work Iten	True
Copy Outputs to Drop Folder	True
Create Work Item on Failure	True
Disable Tests	False
Get Version	
Label Sources	True
MSBuild Arguments	**/p:EnablePackageProcessLoggingAndAssert=true;Dep**
MSBuild Platform	Auto
Private Drop Location	

MSBuild Arguments
Specify any additional command line arguments to pass to MSBuild.exe.

Note: For more information on creating and configuring build definitions, see Creating a Build Definition That Supports Deployment.

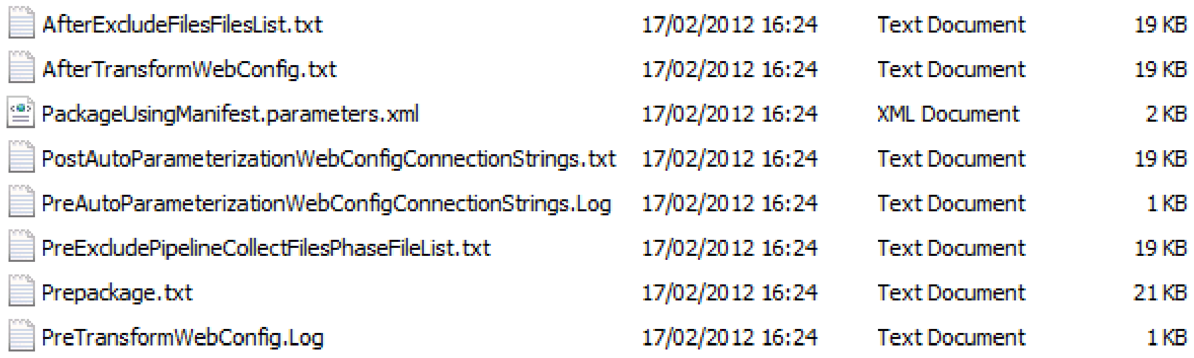
Alternatively, if you want to include the package in every build, you can modify the project file for your web application project to set the **EnablePackageProcessLoggingAndAssert** property to **true**. You should add the property to the first **PropertyGroup** element within your .csproj or .vbproj file.

XML

```xml
<Project ToolsVersion="4.0" DefaultTargets="Build" xmlns="...">
  <PropertyGroup>
    <EnablePackageProcessLoggingAndAssert
       Condition=" '$(EnablePackageProcessLoggingAndAssert)' == '' ">
         true
    </EnablePackageProcessLoggingAndAssert>
    <Configuration Condition=" '$(Configuration)' == '' ">Debug</Configuration>
    <Platform Condition=" '$(Platform)' == '' ">AnyCPU</Platform>
```

Reviewing the Log Files

When you build and package a web application project with **EnablePackageProcessLoggingAndAssert** set to **true**, MSBuild creates an additional folder named Log in the *ProjectName*_Package folder. The Log folder contains various files:



AfterExcludeFilesFilesList.txt	17/02/2012 16:24	Text Document	19 KB
AfterTransformWebConfig.txt	17/02/2012 16:24	Text Document	19 KB
PackageUsingManifest.parameters.xml	17/02/2012 16:24	XML Document	2 KB
PostAutoParameterizationWebConfigConnectionStrings.txt	17/02/2012 16:24	Text Document	19 KB
PreAutoParameterizationWebConfigConnectionStrings.Log	17/02/2012 16:24	Text Document	1 KB
PreExcludePipelineCollectFilesPhaseFileList.txt	17/02/2012 16:24	Text Document	19 KB
Prepackage.txt	17/02/2012 16:24	Text Document	21 KB
PreTransformWebConfig.Log	17/02/2012 16:24	Text Document	1 KB

The list of files that you see will vary according to the things in your project and your build process. However, these files are typically used to record the list of files that the WPP is collecting for packaging, at various stages of the process:

- The *PreExcludePipelineCollectFilesPhaseFileList.txt* file lists the files that MSBuild collects for packaging before any files that are specified for exclusion are removed.
- The *AfterExcludeFilesFilesList.txt* file contains the modified file list after any files that are specified for exclusion are removed.

  Note: For more information on excluding files and folders from the packaging process, see Excluding Files and Folders from Deployment.

- The *AfterTransformWebConfig.txt* file lists the files collected for packaging after any *Web.config* transforms have been performed. In this list, any configuration-specific *Web.config* transform files, like *Web.Debug.config* and *Web.Release.config*, are excluded from the list of files for packaging. A single transformed *Web.config* is included in their place.
- The *PostAutoParameterizationWebConfigConnectionStrings.txt* file contains the list of files after the connection strings in the *Web.config* file have been parameterized. This is the process that lets you replace your connection strings with the right settings for your target environment when you deploy the package.
- The *Prepackage.txt* file contains the finalized pre-build list of files to be included in the package.

Note: The names of the additional log files typically correspond to WPP targets. You can review these targets by examining the *Microsoft.Web.Publishing.targets* file in the %PROGRAMFILES(x86)%\MSBuild\Microsoft\VisualStudio\v10.0\Web folder.

If the contents of your web package aren't what you expected, reviewing these files can be a useful way to identify at what point in the process things went wrong.

Conclusion

This topic described how you can use the **EnablePackageProcessLoggingAndAssert** property in MSBuild to troubleshoot the packaging process. It explained the different ways in which you can supply the property value to the build process, and it described the additional information that is recorded when you set the property to **true**.

Further Reading

For more information on using custom MSBuild project files to control the deployment process, see Understanding the Project File and Understanding the Build Process. For more information on the WPP and how it manages the packaging process, see Building and Packaging Web Application Projects. For guidance on how to exclude specific files and folders from web deployment packages, see Excluding Files and Folders from Deployment.