Sitecore Publishing Service Installation and Configuration Guide

Sitecore XP 8.2

How to install and configure the Sitecore Publishing Service

sitecore®
Own the experience™

Table of Contents

Chapter 1

Introduction

This document describes how to install and configure the Sitecore Publishing Service. It also describes how to install and work with the Sitecore Publishing module.

The document contains the following chapters:

- **Chapter 1 Introduction**
  An introduction to the Sitecore Publishing Service module.
- **Chapter 2 Installing the Sitecore Publishing Service**
  Step-by-step instructions for installing the Sitecore Publishing Service manually or with a script.
- **Chapter 3 Sitecore Publishing Service Commands**
  The various command line arguments and startup modes supported by the Sitecore Publishing Service.
- **Chapter 4 Installing and Configuring the Sitecore Publishing Module**
  Installing the module as a Sitecore package and instructions for post-installation configuration.
- **Chapter 5 Configuring the Sitecore Publishing Service**
  Step-by-step instructions for configuring the Sitecore Publishing Service.
- **Chapter 6 High Availability Configuration of the Sitecore Publishing Service**
  Description on how you can support high availability requirements.
- **Chapter 7 Publishing with the Sitecore Publishing Module**
  Step-by-step instruction on how to use the module to publish Sitecore.
- **Chapter 8 Upgrading from v1.1**
  How to remove the unwanted legacy stored procedures from any databases that you have already deployed.

1.1 About the Publishing Service Module

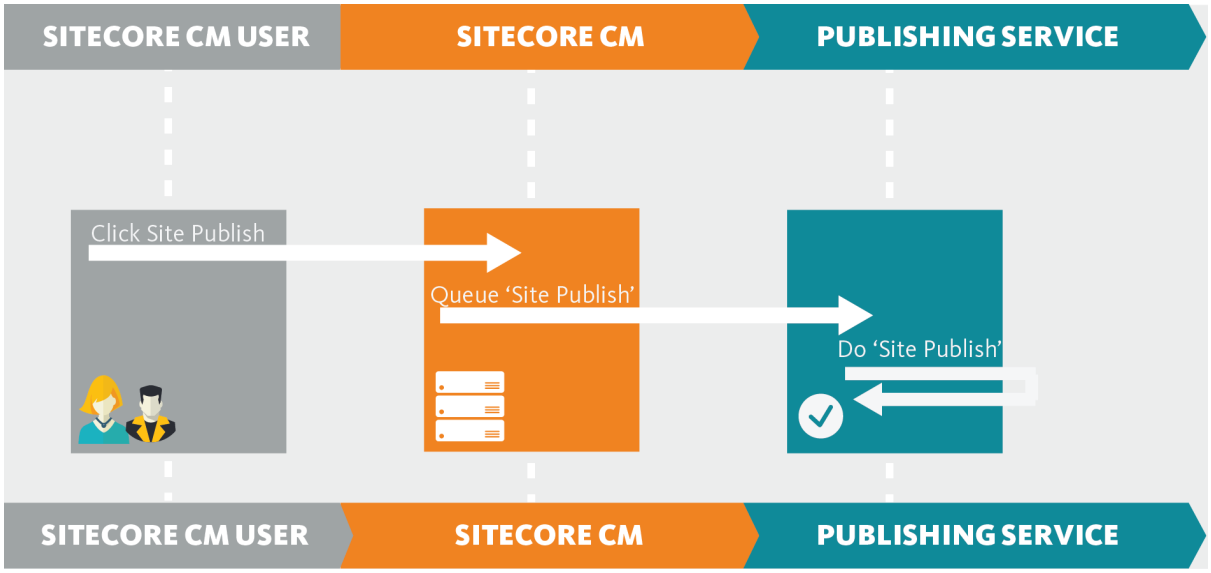
The Publishing Service module is an optional replacement for the existing Sitecore publishing methods. This module increases publishing throughput, reduces the amount of time spent publishing large volumes of items, and offers greater data consistency and reliability. The module also improves the user experience and provides better visual feedback to the user on the state of the publishing system.

The Publishing Service does not use any of the features, pipelines, and settings in the current publishing system. It is an entirely new way of publishing Sitecore items and media.

The Publishing Service runs a separate process to the Sitecore CM instance.

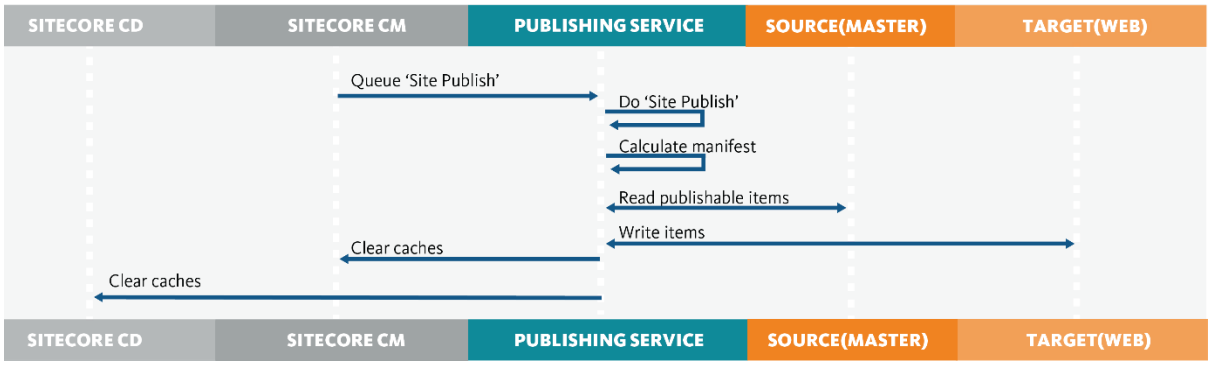
Installation involves:

1. Installation and configuration of the Publishing Service.
2. Installation of the integration module package on your Sitecore instance. The integration module ensures that every publishing action, such as triggering a site publish, is handed on to the publishing service.



When you have installed the Publishing Service, it manages the whole publishing process:

1. It queues and executes publishing jobs.
2. It connects to the Source and Target (SQL) databases directly – reading and writing items in bulk.
3. It issues events, such as cache clearing events, on Content Delivery servers.
4. It reports status information back to UI features, such as the Publishing Dashboard application.



1.1.1 Publishing Service Concepts

The Publishing Service introduces some new concepts for understanding how the different stages of the publishing work are handled:

- **Publishing jobs**

  Previously, when a user chose to publish something, the publishing dialog remained open for the duration of the publish process. This was awkward if the user needed to reboot or if their session ended because they could not see the status of the publishing job.

  The publishing service places all publishing jobs in a queue. When you request a publishing job of any kind, it is queued and then processed as soon as possible. You can see all the active, queued, and completed jobs in the **Publishing Dashboard** application.

- **Manifests**

  This is the collective name for all the tasks that a publishing job performs. The Publishing Service calculates the manifest at the beginning of the publishing job, before it moves any data.

  The Publishing Service looks at the items to see if there any restrictions that would prevent them from being published:

  - Valid dates/workflow states, and so on.
  - Evaluating whether or not the item might need to be deleted.
  - If it is a media file.
  - If extra data needs to be moved along with the item.

  Valid items are added to the manifest as a 'Manifest Step'. Each publishing target gets its own manifest. A publishing job can therefore consist of one or more manifests. The completed manifest is a list of all the items that will be used in the next stage of the process - the Promotion.

- **Promotion**

  This term describes the process of moving the items and data from the source, most often the *Master* database, to one or more publishing targets, such as the *Web* database.

  The Publishing Service creates a manifest and then moves it to one or more publishing targets.

- **Manifest results**

  A list of the changes that were made during the promotion of the manifest. This includes things like item name changes and template updates.

  At the end of the publishing job, the results are passed to the `publishEndResultBatch` pipeline in Sitecore. Developers can hook into this pipeline to work with these results and update any third-party systems or features that may need to know about the changes to items.

  If there is no work to do, that is, if an item is unchanged even though it was in the manifest, a manifest result is not generated.

Chapter 2

Installing the Sitecore Publishing Service

You can install the Sitecore Publishing Service manually or by using the utility scripts that come with the package. This chapter describes:

- Prerequisites
- Manual Installation
- Scripted Installation

2.1 Prerequisites

2.1.1 Sitecore Publishing Service Requirements

The Sitecore Publishing Service comes in a single ZIP archive that you can be execute directly after you have unpacked it. However, you should run the service under IIS because this gives greater configurability of, for example, host addresses and port binding.

- Sitecore Publishing Service 2.0.rev 170130.zip

The prerequisites for the Sitecore Publishing Service 2.0 release are:

- Sitecore XP 8.2 (8.2 rev.160729)
- Windows Server Hosting (.NET Core)

  To enable the service to run under IIS, you must install the Windows Server Hosting package from https://aka.ms/dotnetcore_windowshosting_1_1_0

2.1.2 Sitecore Publishing Module Requirements

The Sitecore Publishing module is distributed as a Sitecore package. This package contains the UI for the publishing service.

- Sitecore Publishing Module 2.0 rev 170130.zip

The prerequisites for the Sitecore Publishing Module 1.8.0 are:

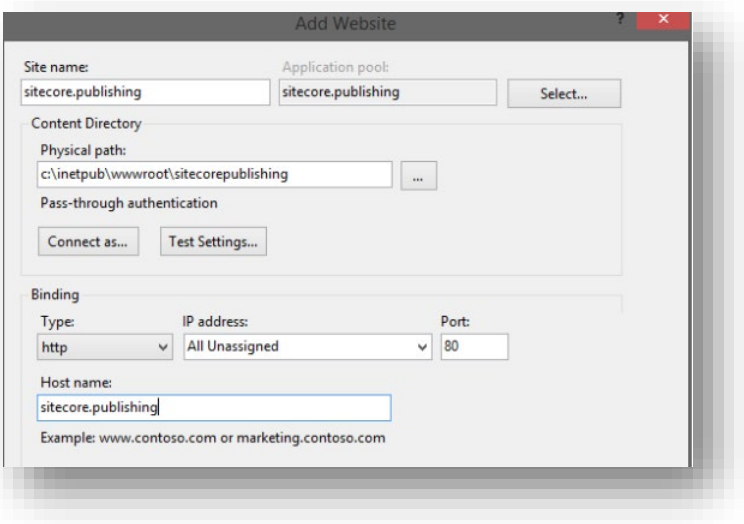
- Sitecore XP 8.2 (8.2 rev.160729)
- The Sitecore Publishing Service 2.0 release

2.2 Manual Installation

Before you install the Sitecore Publishing Service, make sure you have all the prerequisites in place.

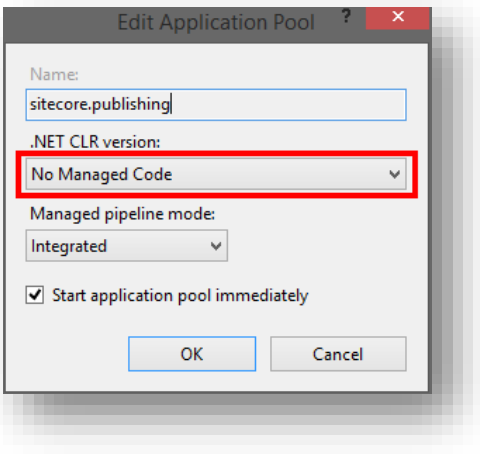
To install the Sitecore Publishing Service for Sitecore 8.2 manually:

1. Download the Sitecore Publishing Service package from the Sitecore Developer Portal.
2. Extract the contents of the archive to a folder of your choice. For example: `C:\inetpub\wwwroot\sitecorepublishing`
3. In IIS, create a new site pointing to the folder.
4. Start the IIS Manager and in the **Connections** panel, expand *Sites*. Right-click *Sites* and then click **Add Website**.
5. In the **Add Website** dialog, fill in the required fields.



Note
If you add a custom host name, you must update your hosts file (`C:\Windows\System32\drivers\etc\`).

6. In the IIS Manager, right-click the application pool for the website that you created, and then click **Basic Settings.**
7. In the **Edit Application Pool** dialog, in the **.NET CLR version** field, select *No Managed Code*.



Note
The Application Pool user must have *Read*, *Execute*, and *Write* permissions to the site's physical path.

8. Configure the connection strings for the service along with any additional configuration values.
9. To upgrade the database schema, run the schema upgrade command from the extracted folder.

   For more information, see the section *Upgrade*.
10. To access your website, enter
    `http://<sitename>/api/publishing/operations/status` in your browser.

    If you receive a value of `{ "Status" : 0 }`, the application is installed correctly. If you receive any other value, check the application logs for further details.

2.3 Scripted Installation

The Sitecore Publishing Service can be installed using commands built in to the application.

To perform a scripted installation:

1. Extract the contents of the archive to a folder of your choice. For example: `c:\inetpub\wwwroot\publishingservice`

   This will be the location where IIS points to the service.

2. To enable the execution of multiple batches on a single connection, configure the connection strings that support Multiple Active Result Sets.

   Note
   If the connection string does not support Multiple Active Result Sets (), it will be changed when you invoke the configuration command.

3. If the provided connection string does not already exist, it will be added to the configuration when you invoke the configuration command. Otherwise, it replaces the connection string with the same key.

   For example, to configure the core, master and web connection strings, run the following commands:

   - `$ Sitecore.Framework.Publishing.Host configuration setconnectionstring core 'value'`
   - `$ Sitecore.Framework.Publishing.Host configuration setconnectionstring master 'value'`
   - `$ Sitecore.Framework.Publishing.Host configuration setconnectionstring web 'value'`

   For more details, see the section *SetConnectionString Command*.

4. Set additional configuration values as needed.

   For example, to set the instance name, run the following command:

   - `$ Sitecore.Framework.Publishing.Host` configuration, set the `Sitecore:Publishing:InstanceName –val MyInstance.`

   For more details, see the section *Set Commands*.

5. Update the relevant schemas.

   For example, to upgrade the schemas to the latest versions, run the schema command:

   - `$ Sitecore.Framework.Publishing.Host schema upgrade –force`

   For more details, see the section *Schema Command*.

6. When the instance is configured and the schemas have been upgraded, you can install it into IIS using the following command:

   - `$ Sitecore.Framework.Publishing.Host iis install –hosts –force`

   For more details, see the section *IIS Command*.

2.3.1 Scaled Environment Considerations

The default configuration for the Publishing Service specifies that the Links Data is stored in the Core database.

If you are running the Publishing Service in a scaled environment and if your Links Data is stored in a different database than the Core database, you must update the Publishing Service configuration accordingly.

For example, if the Links Data is stored in the Web database, then the Publishing Service configuration needs the following override:

```
<DefaultConnectionFactory>
    <Options>
      <Connections>
        <Links>
          <Type>Sitecore.Framework.Publishing.Data.AdoNet.SqlDatabaseConnection,
Sitecore.Framework.Publishing.Data</Type>
          <Options>
            <ConnectionString>${Sitecore:Publishing:ConnectionStrings:Web}</ConnectionString>
          </Options>
        </Links>
```

Chapter 3

Sitecore Publishing Service Commands

This chapter covers the commands that you can use to configure or execute the Sitecore Publishing Service.

This chapter describes:

- Introduction
- Web Command
- IIS Command
- Configuration Command
- Schema Command

3.1 Introduction

The Sitecore Publishing Service supports various command line arguments and startup modes. You can call the application directly to run the default command and optionally pass arguments to modify the execution.

The Web command is the default command for the application. For more details, see the section *Web Command*.

3.1.1 General Execution Format

When you execute the commands, the following applies:

- Executing the .exe will run the default command.
- Options are named, for example, -h, --help. Some of the options may require passing values.
- Arguments are passed separated with a space immediately after the command and before any options.
- Child commands are passed as named arguments immediately after the parent command.

3.1.2 Logs

Any output from a command is added to a *Commands-{data}.log* file in the *Logs* folder in the root of the Publishing Service application.

3.2 Web Command

The Web command is the default command for the application. When the Sitecore Publishing Service starts, it loads the configuration values from the following sources:

- The command line
- The Sitecore configuration files
- The Sitecore environment variables
- The ASPNETCORE environment variables

Note
The configuration values are loaded in the above order, where the values at the command line supersede the others.

The command does not support any specific options or arguments, except from *help* and *version*. However, it does allow the passing of key-value pairs to allow starting the application with different configurations.

You can pass the following options:

Option	Template	Type	Details	Default value
Help	-? | --help	Switch	Displays help information.	
Version	--version	Switch	Displays version information.	
Verbosity	--verbosity	Single - LogLevel	Specify the level at which information is logged to the screen.	Information

3.2.1 Host Configuration Options

To change the startup behavior of the application, you can use the following host configuration options through the command line or as environment variables.

Option	Aspnet Environment	Sitecore Environment	Command line	Type	Details
Detailed Errors	ASPNETCORE_DETAILEDERRORS	SITECORE_DETAILEDERRORS	--detailederrors	Single - Bool	Displays detailed error information instead of generic error pages.
Capture Startup Errors	ASPNETCORE_CAPTURESTARTUPERRORS	SITECORE_CAPTURESTARTUPERRORS	--capturestartuperrors	Single - Bool	Displays errors caused during startup, if possible.
Environment	ASPNETCORE_ENVIRONMENT	SITECORE_ENVIRONMENT	--environment	Single - String	Starts the service in the specified environment.

URLs	ASPNETCORE_URLS	SITECORE_URLS	--urls	Single - String	Starts the service to respond to the specified URLs. Separate multiple URLs with a semicolon, for example, http://localhost:5000;http://localhost:5001.

For example, to start the Publishing Service in a specific environment and on specific URLs:

- `$ Sitecore.Framework.Publishing.Host --urls` 'http://localhost:5000;http://localhost:50001' `--environment Development.`

3.2.2 Custom Configuration Values

Custom configuration values can be passed at the command line or defined via the environment. The values can be set using the following types:

Type	Example
Configuration Key	Sitecore:Publishing:Logging:Filters:Microsoft
Aspnet Environment Variable	Set ASPNETCORE_Sitecore__Publishing__Logging__Filters__Microsoft=Trace
itecore Environment Variable	Set SITECORE_Sitecore__Publishing__Logging__Filters__Microsoft=Trace
Command line	--Sitecore:Publishing:Logging:Filters:Microsoft Trace

When you set the custom configuration values, use the following formats:

- When you set the value as an environment, replace the colon ':' with a double underscore '__'.
- The environment prefix consists of a type (ASPNETCORE or SITECORE) and a single underscore.
- The command line arguments must have the prefix '--'.

3.3 IIS Command

You can install the Publishing Service into the IIS. When you run the command, the site is configured in IIS under the specified sitename and port. The command creates two bindings based on the specified sitename and the machine name and, if requested, it can update the *hosts* file.

When you run the commands, you may receive the following exception:

Exception	Information	Resolution
Cannot read configuration file due to insufficient permissions	One or more IIS configuration files cannot be read by the current user.	Execute the command as a user with the correct permissions.

3.3.1 Install Options

Use the following when you install the Publishing Service on IIS:

Option	Template	Type	Details	Default Value
Help	-? | --help	Switch	Displays help information.	
Version	--version	Switch	Displays version information.	
Verbosity	--verbosity	Single - LogLevel	Specify the level where the information is logged to the screen.	Information
Site Name	-s | --sitename	Single - String	Specify the site that must be installed.	'sitecore.publishing'
App Pool Name	-a | --apppool	Single - String	Specify the application pool for the site.	The sitename
Port Number	-p | --port	Single - Int	Specify the port that must be assigned to the default binding. Must be an integer.	80
Force	--force	Switch	If the site already exists, this switch overwrites the current configuration. Without this, the command fails.	
Hosts	--hosts	Switch	Update the hosts file entry.	

For example:

- To install the service in IIS using the default values:
  - `$ Sitecore.Framework.Publishing.Host iis install`
- To install the service in IIS using specific site and app pool names:
  - `$ Sitecore.Framework.Publishing.Host iis install -site publishing.service -app publishing.service`
- To install the service in IIS using specific site and app pool names, a custom port, and update the machines hosts file (the use of force ensures that any existing site with the same name is updated):
  - `$ Sitecore.Framework.Publishing.Host iis install -site publishing.service -app publishing.service --port 5001 --force –hosts`

3.4 Configuration Command

The configuration command allows configuration values to be persisted in the configuration files for the global or the specific environments.

When you run the commands, you might receive the following exception:

Exception	Information	Resolution
Access to the path '...' is denied	The users do not have access to change the configuration files.	Execute the command as a user with the correct permissions.

3.4.1 Set Commands

With the set command, you can write a configuration value to a configuration file:

Command	Example	Details
Key	`Sitecore:publishing:service:keyname`	Use this to set or modify the configuration file. The command must be separated with a colon ':'.

You can use the following options:

Option	Template	Type	Details	Default value
Help	`-? | --help`	Switch	Displays help information.	
Version	`--version`	Switch	Displays version information	
Verbosity	`--verbosity`	Single - LogLevel	Specify the level at which information is logged to the screen.	Information
Environment	`-e|--environment`	Single - String	Starts the service in the specified environment folder where changes will be persisted.	global
Filename	`-f|--file`	Single - String	Specify the name of the file where changes will be persisted.	sc.custom.json
Value	`-v | --value`	Multiple - String	Specify the value to persist. Repeat use to provide multiple values. If none are provided, 'null' is set as the value or '[]' for arrays.	'null' or '[]'
As Array	`--array`	Switch	Provide this flag to ensure the value is set as an array.	

For example:

- To set a `sitecore:publishing:entry` configuration entry:
  - `$ Sitecore.Framework.Publishing.Host configuration set sitecore:publishing:entry -v myvalue`
- To set the configuration entry in a custom file:
  - `$ Sitecore.Framework.Publishing.Host configuration set sitecore:publishing:entry -v myvalue -f sc.alternate.json`

- To set the configuration entry in an alternative environment:
  - `$ Sitecore.Framework.Publishing.Host configuration set sitecore:publishing:entry -v myvalue -e Development`
- To set the configuration entry to an array of values:
  - `$ Sitecore.Framework.Publishing.Host configuration set sitecore:publishing:entry -v myvalue -v otherValue`
- To set the configuration entry to an array with a single value:
  - `$ Sitecore.Framework.Publishing.Host configuration set sitecore:publishing:entry -v myvalue –array`

3.4.2 SetConnectionString Command

With the SetConnectionString command, you can set or change a connection in a configuration file.

The required arguments are as follows:

Argument	Example	Details
Name	Core	Specify the name of the connection string that you want to configure.
Value	Data Source=.\SQLEXPRESS;Initial Catalog=chromedome_Sitecore.Core;User ID=sa;Password=12345;	Specify the value of the connection string. If the value does not support MARS, it will be updated.

For example:

- To set a connection string value for the Core database:
  - `$ Sitecore.Framework.Publishing.Host configuration setconnectionstring core  Data Source=.\SQLEXPRESS;Initial Catalog=chromedome_Sitecore.Core;User ID=sa;Password=12345;`
- To set the Core database connection string to point to the Master connection string configuration:
  - `$ Sitecore.Framework.Publishing.Host configuration setconnectionstring core {Sitecore:Publishing:ConnectionStrings:Master}`

3.5 Schema Command

With the schema commands, you can install, update, and reset publishing schemas in the databases.

Note
Like the Web command, all the commands allow for the configuration values to be overridden.

You can configure the schemas. For example, to provide alternative connection strings to those in configuration, you can pass them as options.

For example:

- `$ Sitecore.Framework.Publishing.Host schema upgrade –Sitecore:Publishing:ConnectionStrings:Core <corestring> --Sitecore:Publishing:ConnectionStrings:Master <masterstring> --Sitecore:Publishing:ConnectionStrings:Web <webstring>`

When you run the commands, you might receive the following exception:

Exception	Information	Resolution
Create table permission denied in database	The user connecting to the database does not have sufficient permissions.	Provide a connection string with the correct permissions.

3.5.1 Upgrade

Use the following options to upgrade the connections to the specified schemas:

Note
To apply changes, you must use the *--force* flag option.

Option	Template	Type	Details	Default Value
Help	-? | --help	Switch	Displays help information.	
Version	--version	Switch	Displays version information.	
Verbosity	--verbosity	Single - LogLevel	Specify the level at which information is logged to the screen.	Information
Environment	-e|--environment	Single - String	Specify the environment folder to load the connection string configuration.	Production
Schema Version	-sv | --schema-version	Single - Int	Specify the schema version to downgrade to.	0
Force	--force	Switch	Provide this option for the changes to be persisted.	

For example:

- To upgrade the schemas to the latest version:
  - `$ Sitecore.Framework.Publishing.Host schema upgrade –force`
- To upgrade the schemas to version 3:
  - `$ Sitecore.Framework.Publishing.Host schema upgrade -sv 3 –force`

3.5.2 Downgrade Options

Use the following options to downgrade schemas for connections.

Note
To apply changes, you must use the *--force* flag option.

Option	Template	Type	Details	Default Value
Help	-? | --help	Switch	Displays help information.	
Version	--version	Switch	Displays version information.	
Verbosity	--verbosity	Single - LogLevel	Specify the level at which information is logged to the screen.	Information
Environment	-e|--environment	Single - String	Specify the environment folder to load the connection string configuration.	Production
Schema Version	-sv | --schema-version	Single - Int	Specify the schema version to downgrade to.	0
Force	--force	Switch	Provide this option for the changes to be persisted.	

For example:

- To downgrade the schemas to version 0:
  - `$ Sitecore.Framework.Publishing.Host schema downgrade –force`
- To downgrade the schemas to version 3:
  - `$ Sitecore.Framework.Publishing.Host schema downgrade -sv 3 –force`

3.5.3 Reset Options

Use the following options to reset the connections to use the specified schema:

Note
To apply changes, you must use the *--force* flag option.

Option	Template	Type	Details	Default Value
Help	-? | --help	Switch	Displays help information.	
Version	--version	Switch	Displays version information.	
Verbosity	--verbosity	Single - LogLevel	The level at which information is logged to the screen.	Information
Environment	-e|--environment	Single - String	Specify the environment folder to load the connection string configuration.	Production
Schema Version	-sv | --schema-version	Single - Int	Specify the schema version to downgrade to.	0
Force	--force	Switch	Provide this option for the changes to be persisted.	

For example:

- To reset the schemas to the latest version:
  - `$ Sitecore.Framework.Publishing.Host schema reset –force`
- To reset the schemas to version 3:
  - `$ Sitecore.Framework.Publishing.Host schema reset -sv 3 –force`

3.5.4 List

Use the following options to display information for each schema/connection:

Option	Template	Type	Details	Default Value
Help	-? | --help	Switch	Displays help information.	
Version	--version	Switch	Displays version information.	
Verbosity	--verbosity	Single - LogLevel	Specify the level at which information is logged to the screen.	Information
Details	-d | --details	Switch	Display more information for each schema.	
Environment	-e|--environment	Single - String	Specify the environment folder to load the connection string configuration.	Production

For example

- To list detailed information for all schemas:
  - `$ Sitecore.Framework.Publishing.Host schema list –d`
- To list basic information for all schemas in the 'Development' environment:
  - `$ Sitecore.Framework.Publishing.Host schema list -e Development`

Chapter 4

Installing and Configuring the Sitecore Publishing Module

The Sitecore Publishing module is distributed as a standard Sitecore package. However, you must perform some manual configuration steps after the installation.

This chapter describes:

- Installing the Sitecore Publishing Module
- Post-installation Configuration
- Recovery
- Publisher Operations Service
- Security
- Operation Emitter
- Events

4.1 Installing the Sitecore Publishing Module

The Sitecore Publishing module is distributed as a standard Sitecore package. You can install it like any other Sitecore package.

The Sitecore Publishing module package is called:

- Sitecore Publishing Module 2.0 rev 170130.zip

To install the package:

1. On the Sitecore Developer Portal, download the installation package for the module.
2. On the Sitecore Launchpad, click **Control Panel**, and in the **Administration** section, click **Install a package**.

   The **Install a Package** wizard guides you through the installation process.
3. Before you close the wizard, select **Restart the Sitecore Client**.

4.2 Post-installation Configuration

After you have installed the Sitecore Publishing module, you must configure the module before you can use it.

4.2.1 Service Endpoints

To configure the service endpoints:

1. In the *Website root* folder, navigate to the `App_Config/Include/Sitecore.Publishing.Service.Config` configuration file.

   Note
   Do not modify the file directly. Use the Sitecore configuration Include features to change the values.

2. Add a configuration file which overrides the `PublishingServiceUrlRoot` setting to point to your service module:

```
============================================================
***Important! Copy and save this information***
============================================================
    BEFORE YOU CLICK NEXT:
    - Ensure you have installed and configured the Sitecore Publishing
Service (this module only enables integration with the service)
        Documentation detailing how to install the service is available
separately.

        [Warning] This module will not work without a properly configured
service instance. No items will be able to be published.

    AFTER YOU CLOSE THE WIZARD:
    After the package is installed, follow these steps to complete the
Sitecore Publishing Service installation:
    - Configure the service endpoints:
        Add a configuration file which overrides the
'PublishingServiceUrlRoot' setting to point to your service module
        Make sure the address contains a trailing slash
        e.g.
            <?xml version="1.0" encoding="utf-8"?>
            <configuration
xmlns:patch="http://www.sitecore.net/xmlconfig/">
              <sitecore>
                <settings>
                  <setting
name="PublishingServiceUrlRoot">http://sitecore.publishing/</setting>
                </settings>
              </sitecore>
            </configuration>
      - Configure the Content Delivery Servers:
            Rename the file
'Sitecore.Publishing.Service.Delivery.config.disabled' to
            'Sitecore.Publishing.Service.Delivery.config'.
            Ensure that the 'Sitecore.Publishing.Service.Delivery.dll'
exists in the website 'bin' directory
```

Important
Make sure that the URL ends with a trailing slash and that it is formatted correctly.

4.2.2 Content Delivery Servers

To configure the Content Delivery servers to use the publishing service:

1. Enable the `Sitecore.Publishing.Service.Delivery.config.disabled` file by removing the `.disabled` suffix.
2. Ensure that the following files are in the website 'bin' directory:
   - `Sitecore.Framework.Conditions.dll`
   - `Sitecore.Publishing.Service.dll`
   - `Sitecore.Publishing.Service.Abstractions.dll`
   - `Sitecore.Publishing.Service.Delivery.dll`

4.3 Recovery

When an item is modified during service downtime, a recovery process is activated and this process stores any modifications locally. When the service is available again, it recovers the stored changes and pushes them to the service.

The recovery strategy determines how often the recovery process attempts to recover the changes and push them to the publishing service.

To change the frequency of the recovery attempts, change the `interval` setting:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration xmlns:patch="http://www.sitecore.net/xmlconfig/">
  <sitecore>
    <publishing.service>
      <recoveryStrategy>
        <param name="interval">30</param>
      </recoveryStrategy>
    </publishing.service>
  </sitecore>
</configuration>
```

4.4 Publisher Operations Service

To capture the scenarios where the service is down, the `Publisher Operations` service executes requests inside a circuit breaker. When the circuit breaker detects a request error, it tracks the number of failures. When the service detects that the maximum allowed number of failures has been reached, it stops sending requests for a specified period of time.

You can configure the number of failures and the length of the timeout period:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration xmlns:patch="http://www.sitecore.net/xmlconfig/">
  <sitecore>
    <publishing.service>
      <publisherOpsService>
        <param name="circuitbreaker">

          <!-- The number of failed 'add event' requests to the service that are allowed
before determining a communication problem. -->
          <param name="exceptionsAllowedBeforeBreaking">3</param>

          <!-- The duration until communication with the service is attempted again after a
communication problem was last
               detected. -->
          <param name="secondsBeforRetrying">300</param>
        </param>

        <param name="recoveryStrategy" ref="publishing.service/recoveryStrategy"/>
      </publisherOpsService>
    </publishing.service>
  </sitecore>
</configuration>
```

4.5 Security

When you install the Publishing Service module, the *Publishing Service Administrator* role is created. This role has full access to the Publishing Service features, including the full republish functionality on the Publishing Dashboard. The *Sitecore Client Publishing* and the *Sitecore Client Advanced Publishing* roles do not grant access to the full republish functionality.

To avoid security permissions conflicts, you must ensure that users that must be able to perform a full republish are members of the *Publishing Service Administrator* role.

If you are using custom roles, make sure these roles do not inherit permissions from the *Sitecore Client Publishing* or *Sitecore Client Advanced publishing* roles.

- To enable users on your custom roles to perform a full publish, you must edit the configuration file and specifically grant the custom security roles permission to perform a full site publish.

Important
You can only grant permission to perform a full site publish to a security role, not an individual user. Furthermore, you cannot grant permission to perform a full site publish to a security role by only editing the security roles in Sitecore.

4.5.1 Granting Permission to Perform a Full Republish

You can only explicitly grant permission to perform a full site publish to specific security roles in the configuration file. To enable the **Full republish** check box for a role, you must enter the `domain\name` for each role:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration xmlns:patch="http://www.sitecore.net/xmlconfig/">
  <sitecore>
    <publishing.service>
      <api>
        <services>
          <allowFullPublishRoles>
            <role>Publishing Service Administrator</role>
          </allowFullPublishRoles>
        </services>
      </api>
    </publishing.service>
  </sitecore>
</configuration>
```

4.6 Operation Emitter

The operation emitter buffers and streams item changes to the publishing service. You can configure it to stream content more frequently or in larger batches.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration xmlns:patch="http://www.sitecore.net/xmlconfig/">
  <sitecore>
    <publishing.service>
      <operationEmitter>
        <!-- The interval at which an event batch is created to be sent to the service. -->
        <param name="eventBufferingWindowMaxMilliseconds">2000</param>

        <!-- The maximum size of an event batch that will be sent to the service. -->
        <param name="eventBufferingMaxCount">50</param>
      </operationEmitter>
    </publishing.service>
  </sitecore>
</configuration>
```

4.7 Events

The publishing service module exposes a new event in Sitecore:

- publishingservice:publishend

When the publishing service completes a publishing job, this event is triggered once for the entire job in addition to the three events from the existing publishing system:

- publish:begin
- publish:complete
- publish:fail

Chapter 5

Configuring the Sitecore Publishing Service

The Sitecore Publishing Service supports custom configurations.

This chapter contains the following sections:

- Publishing Targets
- Configuration Sources
- Adding Configuration Values
- Overriding Configuration Values
- Referencing Configuration Values
- Configuring Options
- Database Configuration
- Schema Configuration
- Task Scheduling
- Content Availability
- Transient Error Tolerance for SQL Azure
- Logging Configuration
- Troubleshooting

5.1 Publishing Targets

The Publishing Service is configured to use a single publishing target by default, - the Internet.

If you want to publish to another publishing target, you must configure it.

We recommend that you create a patch file to edit the configuration files.

To configure a publishing target:

1. Add the connection string for the new publishing target database to the `ConnectionStrings` section of the configuration

```
<?xml version="1.0" encoding="UTF-8"?>
  <Settings>
    <Sitecore>
      <Publishing>
        <ConnectionStrings>
          <Stage>Data Source=.;Initial Catalog=Preview;Integrated
Security=True;MultipleActiveResultSets=True;ConnectRetryCount=15;ConnectRetryInterval=
1</Stage>
      </Publishing>
    </Sitecore>
  </Settings>
```

2. Add the new publishing target to the `DefaultConnectionFactory` configuration section.

   The name of the XML element in the `DefaultConnectionFactory` section must be the same as the name of the publishing target in Sitecore.

```
<?xml version="1.0" encoding="UTF-8"?>
  <Settings>
    <Sitecore>
      <Publishing>
        <Services>
          <DefaultConnectionFactory>
             <Options>
                <Connections>
                   <Stage> <!—This should be the name of the target in Sitecore ->

<Type>Sitecore.Framework.Publishing.Data.AdoNet.SqlDatabaseConnection,
Sitecore.Framework.Publishing.Data</Type>
                    <LifeTime>Transient</LifeTime>
                    <Options>

<ConnectionString>${Sitecore:Publishing:ConnectionStrings:Stage}</ConnectionString>
                       <DefaultCommandTimeout>120</DefaultCommandTimeout>
                                   <Behaviours>
                                      <backend>sql-backend-default</backend>
                                      <api>sql-api-default</api>
                                   </Behaviours>
                                </Options>
                             </Stage>
                    </Options>
          </DefaultConnectionFactory>
        </Services>
      </Publishing>
    </Sitecore>
  </Settings>
```

3. Add the new publishing target to the StoreFactory configuration section.

```
<?xml version="1.0" encoding="UTF-8"?>
  <Settings>
    <Sitecore>
              <Publishing>
                <Services>
            <StoreFactory>
                       <Options>
                          <Stores>
                             <Targets>
                                <Stage>

<Type>Sitecore.Framework.Publishing.Data.TargetStore,
Sitecore.Framework.Publishing.Data</Type>
                                   <ConnectionName>Stage</ConnectionName>
```

```
<FeaturesListName>TargetStoreFeatures</FeaturesListName>
                                    <Id>GUID FROM SITECORE</Id>
                                    <ScDatabase>Stage</ScDatabase>
                                </Stage>
                            </Targets>
                        </Stores>
                    </Options>
                </StoreFactory>
        </Services>
      </Publishing>
    </Sitecore>
  </Settings>
```

The Id element in the configutation file must be the same as the Sitecore GUID of the publishing target in Sitecore.
The ScDatabase element in the configuration file must be the same as the name of the publishing target item in Sitecore.

5.2 Configuration Sources

During the startup of the Sitecore Publishing Service, the configuration sources are loaded in the following order:

- Environment variables
- Default Sitecore configuration:
  - `<installationPath>\config\sitecore`
- Global configuration:
  - `<installationPath>\config\global`
- Environment specific configuration:
  - `<installationPath>\config\<environment>`

During each stage of the loading, you can override previous values.

Note
If you apply any changes to the configuration files, you must restart the application.

The configuration folder contains all the configuration files for the Publishing Service:

- The *Sitecore* folder contains all the default configuration files provided by Sitecore that you can review to learn what can be configured.

  Important
  Do not modify the files in the *Sitecore* folder. They are automatically overwritten during the upgrade process.

- The *Global* folder – contains the custom/module configuration files that extend or overwrite the Sitecore defaults. This is the location where developers must add their instance specific configuration files, for example, where a configuration file contains details of custom extensions and connection strings.
- <*EnvironmentName*> folder – add custom folders to support different environments. For example, if a *Development* folder exists and the application environment is set to `Development`, the configuration files in this folder are loaded. The default environment `Production` will not load these files.

Configuration File Naming

When you create a configuration file, it must be prefixed with `sc.` in order to be loaded. When you create a configuration file, it must be an `.xml`, `.json`, or `.ini` file in order to be loaded. All other files are ignored.

5.3 Adding Configuration Values

To add a configuration value, declare the value at the relevant path. For example, the default configuration contains an element called <Sitecore><Publishing><ConnectionStrings>:

```
<Settings>
  <Sitecore>
    <Publishing>
      <ConnectionStrings>
        <!-- The Service connection is registered to map to the same connection string as the
master database by default. -->
        <Service>${Sitecore:Publishing:ConnectionStrings:Master}</Service>
      </ConnectionStrings>
      ...
    </Publishing>
  </Sitecore>
</Settings>
```

To add a new value, save the following in:
<installationPath>\config\global\sc.connectionstrings.xml

```
<Settings>
    <Publishing>
      <ConnectionStrings>
        <Master>user id=sa;password=password;data
source=.\SQLEXPRESS;database=sitecore.Master;MultipleActiveResultSets=True;</Master>
      </ConnectionStrings>
    </Publishing>
</Settings>
```

The connection string is now defined at: Sitecore:Publishing:ConnectionStrings:Master

5.4 Overriding Configuration Values

To override a configuration value, you must re-declare the value.

For example, if the default configuration contains an element called `<Sitecore><Publishing><Logging>`:

```
<Sitecore>
    <Publishing>
      <!-- The default Loglevel for the instance. -->
      <Logging>
        <Filters>
          <Sitecore>Information</Sitecore>
        ...
```

Then, you can set the log level to Debug when running in `Development`, by saving the following as: `<installationPath>\config\development\sc.logging.json`

```
{
  "Sitecore": {
    "Publishing": {
      "Logging": {
          "Filters": {
            "Sitecore": "Debug"
           }
        }
    }
  }
}
```

Now, when the Publishing Service starts in a development environment, you get additional logging information.

5.5 Referencing Configuration Values

If you have a configuration value that needs to be referenced elsewhere, you can reference it using the syntax:

- `${ a:b:c }`

This enables you to overwrite the value in a single location, and at the same time the configuration supports its use in multiple configuration files.

For example, the default configuration file contains a connection string entry for the service that is configured to point to the *Master* connection string by default.

If you add a configuration file that contains a value for `Sitecore:Publishing:ConnectionStrings:Master`, the connection string is then used for both the Master database and the Service database.

```
<Settings>
  <Sitecore>
    <Publishing>
      <ConnectionStrings>
        <!-- The Service connection is registered to map to the same connection string as the
master database by default. -->
        <Service>${Sitecore:Publishing:ConnectionStrings:Master}</Service>
      </ConnectionStrings>
    </Publishing>
  </Sitecore>
</Settings>
```

Alternatively, the value at `<Sitecore><Publishing><ConnectionStrings><Service>` could be overwritten in another configuration file that provides an explicit connection string that should be used.

5.6 Configuring Options

You configure the Sitecore Publishing Service by registering object types, so that the service can replace default implementations with custom alternatives. Many of the object types that are registered support an optional configuration section called `Options`.

When an object type supports `Options`, you can provide additional configuration values to change the behavior of the application.

5.6.1 DatabaseConnectionOptions

You can use the `DatabaseConnectionOptions` class to specify the connection to a data source.

The `DatabaseConnectionOptions` class is used by the type:

- `Sitecore.Framework.Publishing.Data.AdoNet.SqlDatabaseConnection`.

```
namespace Sitecore.Framework.Publishing.Data.AdoNet
{
    public class DatabaseConnectionOptions
    {
        public string ConnectionString { get; set; }

        public int CommandTimeout { get; set; } = 120;

       public Dictionary<string, string> Behaviours { get; set; } = new Dictionary<string,
string>(StringComparer.OrdinalIgnoreCase);
    }
}
```

The following example specifies an alternative value for the `CommandTimeout` setting of the Service connection:

```
<Sitecore>
    <Publishing>
      <Services>
        <DefaultConnectionFactory>
          <Options>
            <Service>
              <Options>
                <CommandTimeout>30</CommandTimeout>
              </Options>
            </Service>
          </Options>
        </DefaultConnectionFactory>
      </Services>
    </Publishing>
  </Sitecore>
</Settings>
```

5.6.2 PublishHostOptions

You can use the `PublishHostOptions` class to specify the main configuration options for logging in the Publishing Service and to specify the collection of services that must be registered. Services are all the types that are registered during start up.

The `PublishHostOptions` class is used by the type:

- `Sitecore.Framework.Publishing.Host`.

```
namespace Sitecore.Framework.Publishing.Host
{
    public class PublishHostOptions
    {
        public List<ConfigurationServiceType> Services { get; set; } = new
List<ConfigurationServiceType>();

        public LoggingHostOptions Logging { get; set; }
    }
}
```

In the following example, a custom service is added to the collection of services.

```
<Settings>
  <Sitecore>
    <Publishing>
      <Services>
        <MyCustomService>
          <Type>MyCustom.Service, MyCustom</Type>
          <As>MyCustom.IService, MyCustom.Abstractions</As>
        </MyCustomService>
      </Services>
    </Publishing>
  </Sitecore>
</Settings>
```

Note
To change the logging setup, see the section Logging Configuration.

5.6.3 PublishJobHandlerOptions

You can use the `PublishJobHandlerOptions` class to configure various aspects of the Publish Job handler implementations to optimize performance.

The `PublishJobHandlerOptions` class is used by the type:

- `Sitecore.Framework.Publishing.PublishJobQueue.Handlers.IncrementalPublishHandler`

```
namespace Sitecore.Framework.Publishing.PublishJobQueue
{
    public class PublishJobHandlerOptions
    {
        public int RelatedItemBatchSize { get; set; } = 2000;

        public int ManifestBuilderBatchSize { get; set; } = 5000;

        public int UnpublishedOperationsLoadingBatchSize { get; set; } = 2000;

         public int DeletedItemsBatchSize { get; set; } = 2000;

        public int MediaBatchSize { get; set; } = 2000;

        public int IndexReadBatchSize { get; set; } = 500;

        public int IndexWriteBatchSize { get; set; } = 500;

        public int TargetOperationsBatchSize { get; set; } = 2000;

        public int SourceTreeReaderBatchSize { get; set; } = 2000;

        public bool TransactionalPromote { get; set; } = true;

        public bool ParallelPromote { get; set; } = true;

        public bool ContentTesting { get; set; } = true;

  public bool ContentAvailability { get; set; } = false;
    }
}
```

The following configuration example specifies an alternative value for the default configuration:

```
<Settings>
  <Sitecore>
    <Publishing>
      <Services>
        <IncrementalPublishHandler>
          <Options>
            <ContentTesting>True</ContentTesting>
            <IndexReadBatchSize>500</IndexReadBatchSize>
            <IndexWriteBatchSize>500</IndexWriteBatchSize>
            <ManifestBuilderBatchSize>5000</ManifestBuilderBatchSize>
            <ParallelPromote>False</ParallelPromote>
```

```
            <RelatedItemBatchSize>2000</RelatedItemBatchSize>
            <SourceTreeReaderBatchSize>2000</SourceTreeReaderBatchSize>
            <TargetOperationsBatchSize>2000</TargetOperationsBatchSize>
            <TransactionalPromote>True</TransactionalPromote>

<UnpublishedOperationsLoadingBatchSize>2000</UnpublishedOperationsLoadingBatchSize>
          </Options>
        </IncrementalPublishHandler>
      </Services>
    </Publishing>
  </Sitecore>
</Settings>
```

5.6.4 PromoterOptions

You use the `PromoterOptions` class to configure various aspects of the Publish Job promoter implementations to optimize performance.

The `PromoterOptions` class is used by:

- `Sitecore.Framework.Publishing.DataPromotion.DefaultItemCloneManifestPromoter`
- `Sitecore.Framework.Publishing.DataPromotion.DefaultItemManifestPromoter`
- `Sitecore.Framework.Publishing.DataPromotionDefaultMediaManifestPromoter`

```
namespace Sitecore.Framework.Publishing.Abstractions.DataPromotion
{
    public class PromoterOptions
    {
        public int BatchSize { get; set; } = 500;
    }
}
```

The following configuration example specifies an alternative `BatchSize` class for the registered `ItemCloneManifestPromoter`:

```
<Settings>
  <Sitecore>
    <Publishing>
      <Services>
        <ItemCloneManifestPromoter>
          <Options>
            <BatchSize>1000</BatchSize>
          </Options>
        </ItemCloneManifestPromoter>
      </Services>
    </Publishing>
  </Sitecore>
</Settings>
```

5.6.5 PromotionCoordinatorOptions

You can use the `DefaultPromotionCoordinatorOptions` class to specify whether the Publishing Service should update the *Descendants* table. The default value is `false`, as this improves the performance of the publishing process.

The `DefaultPromotionCoordinatorOptions` class is used by the type:

- `Sitecore.Framework.Publishing.DataPromotion.DefaultPromotionCoordinator`

```
namespace Sitecore.Framework.Publishing.DataPromotion
{
    public class DefaultPromotionCoordinatorOptions
    {
        public bool RebuildDescendantsTable { get; set; } = false;
```

```
    {
}
```

The following configuration example illustrates how to update the *Descendants* table:

```
<Settings>
  <Sitecore>
    <Publishing>
      <Services>
        <PromotionCoordinator>
          <Options>
            <RebuildDescendantsTable>true</RebuildDescendantsTable>
          </Options>
        </PromotionCoordinator>
      </Services>
    </Publishing>
  </Sitecore>
</Settings>
```

Note
If you use Sitecore's digital marketing functionality, you must update the *Descendants* table.

5.7 Database Configuration

Database configuration details can be seen in the `sc.publishing.xml` configuration file.

For SQL database connections, the user defined in the connection string must have the following permissions:

- Delete
- Execute
- Insert
- Select
- Update

Note
In addition, for executing the schema commands, the user must also have the *Alter* permission.

5.7.1 Connection Strings

The connection strings are configured under `<Sitecore><Publishing><ConnectionStrings>`.

Sitecore expects three default connection strings to be configured – *core*, *web*, and *master,* and these are referenced elsewhere in the configuration.

```
<Settings>
    <Publishing>
      <ConnectionStrings>
        <Master>user id=sa;password=password;data
source=.\SQLEXPRESS;database=sitecore.Master;MultipleActiveResultSets=True;</Master>
        <Web>user id=sa;password=password;data
source=.\SQLEXPRESS;database=sitecore.Web;MultipleActiveResultSets=True;</Web>
        <Core>user id=sa;password=password;data
source=.\SQLEXPRESS;database=sitecore.Core;MultipleActiveResultSets=True;</Core>
      </ConnectionStrings>
    </Publishing>
</Settings>
```

Currently, SQL connection strings require that they support Multiple Active Result Sets (MARS), so when configuring a connection string, you must set `MultipleActiveResultSets` to `true`.

Use the following format or similar for connection strings:

- `Data Source = .\\SQLEXPRESS; Initial Catalog = publishing_master; Integrated Security=True; MultipleActiveResultSets=True;`

For more information, see https://www.connectionstrings.com/sqlconnection/.

5.7.2 DefaultConnectionFactory

In the `DefaultConnectionFactory` configuration, the connections are defined. Each connection defines its type, configuration options, and name.

The following example defines a connection called `Internet` that uses the *web* connection string:

```
<DefaultConnectionFactory>
  <Options>
    <Connections>
      <Internet>
        <!-- Should match the name of the publishing target configured in SC. -->
        <Type>Sitecore.Framework.Publishing.Data.AdoNet.SqlDatabaseConnection,
Sitecore.Framework.Publishing.Data</Type>
        <LifeTime>Transient</LifeTime>
        <Options>
          <ConnectionString>${Sitecore:Publishing:ConnectionStrings:Web}</ConnectionString>
          <DefaultCommandTimeout>120</DefaultCommandTimeout>
          <Behaviours>
            <backend>sql-backend-default</backend>
```

```
            <api>sql-api-default</api>
          </Behaviours>
        </Options>
      </Internet>
    </Connections>
  </Options>
</DefaultConnectionFactory>
```

The following connections are configured by default:

Connections	Type	Points to
Links	SQL	Core connection string
Service	SQL	Service connection string
Master	SQL	Master connection string
Internet	SQL	Web connection string

5.7.3 StoreFactory

The `StoreFactory` configuration configures stores in the application that binds one or more connections to a collection of features.

The configuration of Stores is divided into the following sections:

Store Type	Connections	Details
Service	Service	The store containing service data.
Sources	Master	The store(s) for source data. Each source can register multiple connections.
Targets	Internet	The store(s) for target data. Each entry is a possible publish target.
ItemsRelationship	Links	The store for relationship information.
Custom	User defined	Optional custom data stores can be configured.

The following example defines the Sources and Targets sections:

```
<StoreFactory>
   <Options>
     <Stores>
       <Sources>
         <Master>
           <Type>Sitecore.Framework.Publishing.Data.SourceStore,
Sitecore.Framework.Publishing.Data</Type>
           <ConnectionNames>
             <master>Master</master>
           </ConnectionNames>
           <FeaturesListName>SourceStoreFeatures</FeaturesListName>
           <!-- The name of the Database entity in Sitecore. -->
           <ScDatabase>master</ScDatabase>
         </Master>
       </Sources>
       <Targets>
         <!--Additional targets can be configured here-->
         <Internet>
           <Type>Sitecore.Framework.Publishing.Data.TargetStore,
Sitecore.Framework.Publishing.Data</Type>
           <ConnectionName>Internet</ConnectionName>
           <FeaturesListName>TargetStoreFeatures</FeaturesListName>
           <!-- The id of the target item definition in Sitecore. -->
           <Id>8E080626-DDC3-4EF4-A1D1-F0BE4A200254</Id>
           <!-- The name of the Database entity in Sitecore. -->
           <ScDatabase>web</ScDatabase>
         </Internet>
       </Targets>
```

```
      </Stores>
    </Options>
</StoreFactory>
```

Note
The Sources and Targets must set the `ScDatabase` property. Targets must also set the `Id` property.

5.7.4 StoreFeatureLists

The `StoreFeatureLists` configuration specifies the list of features that are available on a particular store.

In the following example, the features that are available to the source store are a number of repositories. A store feature list is linked back to a store via its name that is stored in the `FeatureListName` element.

```
<StoreFeaturesLists>
          <Options>
            <FeatureLists>
              <!--Source Store Features-->
              <SourceStoreFeatures>
                <ItemReadRepositoryFeature>
                  <Type>Sitecore.Framework.Publishing.Data.CompositeItemReadRepository,
Sitecore.Framework.Publishing.Data</Type>
                </ItemReadRepositoryFeature>
                <TestableContentRepositoryFeature>
                  <Type>Sitecore.Framework.Publishing.Data.CompositeTestableContentRepository,
Sitecore.Framework.Publishing.Data</Type>
                </TestableContentRepositoryFeature>
                <WorkflowStateRepositoryFeature>
                  <Type>Sitecore.Framework.Publishing.Data.CompositeWorkflowStateRepository,
Sitecore.Framework.Publishing.Data</Type>
                </WorkflowStateRepositoryFeature>
                <EventQueueRepositoryFeature>
                  <Type>Sitecore.Framework.Publishing.Data.CompositeEventQueueRepository,
Sitecore.Framework.Publishing.Data</Type>
                  <options>
                    <ConnectionName>master</ConnectionName>
                  </options>
                </EventQueueRepositoryFeature>
              </SourceStoreFeatures>
            </FeatureLists>
          </Options>
</StoreFeaturesLists>
```

5.7.5 Custom Data Providers

To support multiple providers of data for a source store, you can add custom data providers to the system.

To add custom data providers to the system:

1. Create a class that implements the `IIndexableItemReadRepository` interface. The following three methods are contained with the type:
   - `GetItemNodeDescriptors` – this method must be implemented to return all the items contained within the custom data provider. The `IItemNodeDescriptor` interface only contains a small number of properties to represent each item.
   - `GetItemNodes` – this method returns `IEnumerable<IItemNode>` when a list of item Guids is supplied. The `IItemNode` represents an item including its field data.
   - `GetVariants` – this method returns a `IEnumerable<IItemVariant>` when supplied with a list of `IDataLocators`. The `IItemVarient` represents an item variant (language and version) and its corresponding fields.
2. Create a connection class. You can inherit from `IConnection`, or use an existing type (for example, `SQLDatabaseConnection`).

3. Create a repository builder by implementing `DefaultRepositoryBuilder<IItemReadRepository, TRepo, TConnection>`, where:
   - `TRepo` is what you entered in step 1
   - `TConnection` is what you entered in step 2.
4. Update the configuration:

```
<Settings>
  <Sitecore>
    <Publishing>
      <Services>
        <!-- Register the custom repository builder -->
        <MyCustomItemReadRepositoryBuilder>
          <Type>My.Custom.ItemReadRepositoryBuilder, My.Custom</Type>
          <As>Sitecore.Framework.Publishing.Repository.IRepositoryBuilder`1[[Sitecore.Framewor
k.Publishing.Item.IIndexableItemReadRepository,
Sitecore.Framework.Publishing.Service.Abstractions]],
Sitecore.Framework.Publishing.Service.Abstractions</As>
        </MyCustomItemReadRepositoryBuilder>
        <DefaultConnectionFactory>
          <Options>
            <Connections>
              <!-- Register the custom connection -->
              <Custom>
                <Type>My.Custom.FileSystemProvider.FileSystemConnection, My.Custom</Type>
                <Lifetime>Transient</Lifetime>
                <Options>
                  <IdTablePrefix>pubExample</IdTablePrefix>
                  <IdTableConnection>Master</IdTableConnection>
                  <RootFolder>C:\siecoredata\Data\CustomItems</RootFolder>
                </Options>
              </Custom>
            </Connections>
          </Options>
      </DefaultConnectionFactory>
        <StoreFactory>
          <Options>
            <Stores>
              <Sources>
                <Master>
                 <!-- add the connection to the master source -->
                  <ConnectionNames>
                  <custom>Custom</custom>
                </ConnectionNames>
              </Master>
            </Sources>
          </Stores>
        </Options>
      </StoreFactory>
      </Services>
    </Publishing>
  </Sitecore>
</Settings>
```

Note
Currently, the publishing service supports reading from a custom data provider, for example, reading from a customized source and then publishing that data as Sitecore items to the target database.

However, writing or publishing to a custom data provider is not currently supported.

5.8 Schema Configuration

During startup, the Sitecore Publishing Service checks whether the latest version of the schema is installed. If the schema needs to be updated, the service shuts down.

You can use the schema command to update and install schemas in the registered connections.

A schema is defined as a DLL that contains a set of resources for preparing a connection for its role in the service. The resources are organized into versions to support incremental schema upgrade and downgrade. This means that, in the example of an SQL schema, the DLL contains multiple scripts for dropping and recreating tables, stored procedures, and other requirements for accessing SQL data.

Schemas can be split, based on their feature set and/or their connection type and they are configured under `<Sitecore><Publishing><Services><SchemaInstaller><Options>`.

The following code sample is the default schema configuration that defines the suite of schemas that are installed by the schema update tool:

```
<Settings>
  <Sitecore>
    <Publishing>
      <Services>
        <SchemaInstaller>
          <Options>
            <!--
              The DeploymentMap defines which schemas are loaded into which connection
            -->
            <DeploymentMap>
              <Custom>
                <Links>
                  <Common>Common</Common>
                  <Data-Common>Data-Common</Data-Common>
                  <Data-Links>Data-Links</Data-Links>
                </Links>
              </Custom>
              <Service>
                <Common>Common</Common>
                <Service>Service</Service>
              </Service>
              <Source>
                <Common>Common</Common>
                <Data-Common>Data-Common</Data-Common>
                <Data-Source>Data-Source</Data-Source>
              </Source>
              <Target>
                <Common>Common</Common>
                <Data-Common>Data-Common</Data-Common>
                <Data-Target>Data-Target</Data-Target>
              </Target>
            </DeploymentMap>
            <!--
              The schemas bind names from the DeploymentMap to a Type/Assembly containing sql
schemas to be loaded
            -->
            <Schemas>
              <Common>Sitecore.Framework.Publishing.Common.Sql.Schema,
Sitecore.Framework.Publishing.Common.Sql.Schema</Common>
              <Data-Common>Sitecore.Framework.Publishing.Data.Common.Sql.Schema,
Sitecore.Framework.Publishing.Data.Common.Sql.Schema</Data-Common>
              <Data-Links>Sitecore.Framework.Publishing.Data.Links.Sql.Schema,
Sitecore.Framework.Publishing.Data.Links.Sql.Schema</Data-Links>
              <Data-Source>Sitecore.Framework.Publishing.Data.Source.Sql.Schema,
Sitecore.Framework.Publishing.Data.Source.Sql.Schema</Data-Source>
              <Data-Target>Sitecore.Framework.Publishing.Data.Target.Sql.Schema,
Sitecore.Framework.Publishing.Data.Target.Sql.Schema</Data-Target>
              <Service>Sitecore.Framework.Publishing.Service.Sql.Schema,
Sitecore.Framework.Publishing.Service.Sql.Schema</Service>
            </Schemas>
          </Options>
        </SchemaInstaller>
      </Services>
    </Publishing>
  </Sitecore>
</Settings>
```

5.8.1 The Deployment Map

The `DeploymentMap` section maps the schemas to connection types.

The following code sample binds the *Common*, *Data-Common*, and *Data-Links* schemas that must be installed on the custom *Links* connection. The *Common* and *Service* schemas are installed on the *Service* connection.

```
<DeploymentMap>
              <Custom>
                <Links>
                  <Common>Common</Common>
                  <Data-Common>Data-Common</Data-Common>
                  <Data-Links>Data-Links</Data-Links>
                </Links>
              </Custom>
              <Service>
                <Common>Common</Common>
                <Service>Service</Service>
              </Service>
```

5.8.2 Schemas

The `Schemas` section names all of the schemas that are installed.

Each configuration value should point to a type in an assembly where the schemas can be discovered. The following code sample names the `Sitecore.Framework.Publishing.Common.Sql.Schema` assembly as `Common` and the `Sitecore.Framework.Publishing.Data.Common.Sql.Schema` assembly as `Data-Common`:

```
<Schemas>
    <Common>Sitecore.Framework.Publishing.Common.Sql.Schema,
Sitecore.Framework.Publishing.Common.Sql.Schema</Common>
    <Data-Common>Sitecore.Framework.Publishing.Data.Common.Sql.Schema,
Sitecore.Framework.Publishing.Data.Common.Sql.Schema</Data-Common>
```

5.8.3 Validating Schemas

When the publishing service starts, it checks whether the latest schema is installed. The version of the installed schemas retrieved from the `__PublishingSchema` table is compared to the schema version in the resource file. If a schema upgrade is needed, the service will shut down and log an error message telling you to upgrade the schema.

5.9 Task Scheduling

The task scheduler is a service that manages the creation of tasks at start up as well as enabling the addition and execution of tasks at runtime.

5.9.1 Task Configuration

The Publishing Service enables you to configure independent tasks in the system. It contains two task definitions by default:

- `PublishTask` – the task that handles requests to publish items from sources to targets.
- `PublishJobCleanUpTask` – the tasks that handles the periodic clean-up of historical publishing jobs.

The default task configuration is contained in the `config\sitecore\sc.publishing.tasks.xml` configuration file.

PublishTask

The `PublishTask` task definition is configured with two triggers:

- `Interval` – the interval trigger runs every few seconds to check for publishing jobs that were requested while the previous publishing job was running.
- `Event` – the event-based trigger causes a publishing job to start immediately after it is requested. If a publishing job is already being processed, the job is delayed until the next interval.

PublishJobCleanUpTask

The `PublishJobCleanUpTask` task definition removes old publishing jobs from the database to prevent the buildup of data over time. It has a single trigger raising on an infrequent schedule to remove jobs over a week old.

You can configure the task by changing its options:

- `JobAge` – the time that must have passed since a publishing job's `Stopped` time. The default value is seven days. If a publishing job's `Stopped` time is older than the `JobAge`, it is eligible for clean-up.
- BatchSize – this is the number of items in the batch that can be deleted together. The default value is 50.

5.9.2 Defining a Task

When you have implemented a task, it must be added to the configuration so that it can be created at startup:

```
<Settings>
  <Sitecore>
    <Publishing>
      <Services>
        <Scheduler>
          <Options>
            <Tasks>
              <CustomTask>
                <TaskDefinition Type="Custom.Task, Custom" BindOptions="property">
                  <Options>
                    <Id>Custom Task</Id>
                    <Categories>
                      <Custom>Custom</Custom>
                      <Other>Other</Other>
                    </Categories>
                  </Options>
                </TaskDefinition>
              </CustomTask>
            </Tasks>
          </Options>
        </Scheduler>
      </Services>
```

```
    </Publishing>
  </Sitecore>
</Settings>
```

A task can expose additional parameters, such as `ID` and `Categories`, to help identify the task when the system is running.

5.9.3 Defining a Trigger

A task cannot run if there are no triggers associated with it. Each trigger is a unique instance, so you can register multiple triggers of the same type. For example, two interval triggers could be registered that trigger a task at different polling intervals:

```
<Settings>
  <Sitecore>
    <Publishing>
      <Services>
        <Scheduler>
          <Options>
            <Tasks>
              <CustomTask>
                <TaskDefinition Type="Custom.Task, Custom" BindOptions="property">
                  <Options>
                    <Id>Custom Task</Id>
                    <Categories>
                      <Custom>Custom</Custom>
                      <Other>Other</Other>
                    </Categories>
                  </Options>
                </TaskDefinition>
                <TriggerDefinitions>
                  <Interval1
Type="Sitecore.Framework.Scheduling.Triggers.IntervalTriggerDefinition,
Sitecore.Framework.Scheduling" BindOptions="property">
                    <Options Interval="00:10:00" /> <!-- Raise every ten minutes -->
                  </Interval1>
                  <Interval2
Type="Sitecore.Framework.Scheduling.Triggers.IntervalTriggerDefinition,
Sitecore.Framework.Scheduling" BindOptions="property">
                    <Options Interval="00:00:10" /> <!-- Raise every ten seconds -->
                  </Interval2>
                </TriggerDefinitions>
              </CustomTask>
            </Tasks>
          </Options>
        </Scheduler>
      </Services>
    </Publishing>
  </Sitecore>
</Settings>
```

5.10 Content Availability

The content availability feature ensures that the valid version of an item is always available in the target database at the time of publishing. In this way, you do not have to perform a publishing operation every time an item version expires and the next version should be displayed.

When you enable content availability and publish an item, the currently valid item version and all the versions that are valid for future publishing are moved from the source database to the target database. A new pipeline that is enabled in the content availability configuration file automatically clears the Sitecore item cache when an item version expires and then, when a contact accesses the item, the next valid version is displayed.

Important
If you are using HTML caching on a rendering, the Sitecore item cache does not automatically clear. In addition, if you use a data source inside a rendering, and if the data source item switches to display a new version, the hosting rendering is not updated because there is nothing that indicates that an update of a dependant data source has been triggered.

5.10.1 Configure Content Availability on the CD Server

To configure content availability on the Content Delivery (CD) server:

1. Place the DLL *Sitecore.Publishing.Service.Delivery.dll* in the bin directory of the CD server.
2. Copy the *Sitecore.Publishing.Service.ContentAvailability.config* file to the CD server and enable it.
3. If you use Lucene for content search, enable the *Sitecore.Publishing.Service.ContentAvailability.lucene.config* file.
4. If you use Solr for content search, enable the *Sitecore.Publishing.Service.ContentAvailability.solr.config* file.
5. Restart your instance.

When an item is indexed the computed fields below stores the valid inception and expiry dates for each version. When a query is issued to content search then the *isAvailable* flag is checked to ensure the hiding and display of the valid versions in a search context that matches the behaviour of the Item API.

The content availability functionality adds the following new fields:

- Computed fields:
  - versionsunrisedate
  - versionsunsetdate
  - publishablefrom
  - publishableto
- Virtual field
  - isAvailable

Publishing Service Setup

To enable content availability in the Publishing Service:

1. In the `config` directory of the Publishing Service, enable the *sc.publishing.contentavailability.xml* file.
2. Restart the Publishing Service.
3. With DEBUG logging enabled, ensure that the Content Availability status is set to ON.

When the content availability is enabled, the:

- `Filter items` pipeline enables publication checks on items as they come out of the database.
- GetLinqFilter processor and VirtualField amend a publication check to each LINQ query going out so that non-published data does not show.

Important
It is possible to misconfigure an items validity period so that it becomes invalid and disappears. For example, if you set the **PublishFrom** field to *02nd January 2017* and the **PublishTo** field to *1st January 2017*, the item does not have a valid date range that allows the item to be displayed. In Content Editor, in the Publishing Viewer, you can to see a visual representation of the date range of an item or item version and diagnose these sorts of errors.

5.11 Transient Error Tolerance for SQL Azure

If you host any application databases in SQL Azure, Microsoft recommends that you implement a retry strategy for all the database requests to overcome any transient errors that might occur due to the nature of a shared cloud infrastructure.

Note
For an introduction to transient errors in SQL Azure, see:
http://social.technet.microsoft.com/wiki/contents/articles/18665.cloud-service-fundamentals-data-access-layer-transient-fault-handling.aspx#Timeouts_amp_Connection_Management.

The Publishing Service provides an implementation of this retry behavior for ADO.NET database requests, however, you must explicitly enable the behavior via configuration according to which databases are hosted on SQL Azure:

- The retry behavior is defined in …`\config\azure\sc.publishing.sqlazure.xml`.
- A typical configuration setup is provided with the Publishing Service in …`\config\azure\sc.publishing.sqlazure.connections.xml.example`. Edit this file accordingly and enable it by removing the `.example` extension.

  For more information about editing this file, see the section *SQL Azure Configuration*.

Because both files are supplied in the *Azure environment* folder, you must start the service with the `environment` setting set to Azure. You can move these files into a different environment folder to achieve a different behavior.

5.11.1 Connection Behaviors

By default, the Publishing Service comes with the concept of connection behaviors that provide the opportunity for transient errors to be mitigated seamlessly in the application for ADO.NET connections.

When submitting a request to the database in the Publishing Service with ADO.NET, a connection behavior is chosen according to the connection used and the context in which the request is made.

The context is a Data Access Context, which is either *api* or *backend*, depending on the type of work that is performed in each part of the system:

- api – when the data is being processed to serve a request for information from an out-of-process component (for example, the publishing service API).
- backend – when data is being processed as part of a background operation (for example, a publishing job).

Note
Microsoft recommends that you configure the API and Backend contexts differently with regards to transient error handling.

A connection behavior is essentially a component that can wrap each command sent to the database, and thereby catch any exceptions that get returned, and repeat the command any number of times if necessary.

By default, the Publishing Service is configured with a *no retry* connection behavior for all connections and contexts, which is essentially a *null* behavior that does not provide any additional logic.

5.11.2 Default Configuration

The connection behaviors are configured in the `Settings\Sitecore\Services\DbConnectionBehaviours` section of the configuration. The connection behavior used when a request is made to a database is chosen according to the current Data Access Context, and the name of the connection behavior configured for the current connection.

The following sample is an extract from the default configuration for the Service connection, where you can see that the Service connection is configured to use the *sql-backend-default* and *sql-api-default* behaviors for the *api* and *backend* contexts respectively.

```
<Service>
  <Type>Sitecore.Framework.Publishing.Data.AdoNet.SqlDatabaseConnection,
Sitecore.Framework.Publishing.Data</Type>
  <LifeTime>Transient</LifeTime>
  <Options>
    <ConnectionString>${Sitecore:Publishing:ConnectionStrings:Service}</ConnectionString>
     <DefaultCommandTimeout>120</DefaultCommandTimeout>
     <Behaviours>
         <backend>sql-backend-default</backend>
          <api>sql-api-default</api>
     </Behaviours>
  </Options>
</Service>
```

The following sample is an extract from the default configuration of the two connection behaviors. This configuration results in all connections having no special behavior around connections, and enables you to configure different command timeouts for the different contexts:

```
<DbConnectionBehaviours>
          <Options>
            <Entries>
              <!-- Used for all DatabaseConnections created in backend contexts (typically
publishing jobs). -->
              <sql-backend-default>

<Type>Sitecore.Framework.Publishing.Data.AdoNet.NoRetryConnectionBehaviour,
Sitecore.Framework.Publishing.Data</Type>
                <Options>
                  <Name>Default Backend No Retry behaviour</Name>
                  <CommandTimeout>120</CommandTimeout>
                </Options>
              </sql-backend-default>

              <!-- Used for all DatabaseConnections created in API contexts. -->
              <sql-api-default>
                <Type>Sitecore.Framework.Publishing.Data.AdoNet.NoRetryConnectionBehaviour,
Sitecore.Framework.Publishing.Data</Type>
                <Options>
                  <Name>Default Api No Retry behaviour</Name>
                  <CommandTimeout>10</CommandTimeout>
                </Options>
              </sql-api-default>

            </Entries>
          </Options>
</DbConnectionBehaviours>
```

5.11.3 SQL Azure Configuration

The connection behaviors in the Publishing Service are aligned with the recommendations from Microsoft on mitigating transient errors in SQL Azure. They are specified in the …\config\azure\sc.publishing.sqlazure.xml file.

In this file, the following two connection behaviors are added:

```
<sql-backend-azure>

<Type>Sitecore.Framework.Publishing.Data.AdoNet.ExponentialBackoffDbConnectionBehaviour`1[[Mic
rosoft.Practices.EnterpriseLibrary.TransientFaultHandling.SqlDatabaseTransientErrorDetectionSt
rategy, Microsoft.Practices.EnterpriseLibrary.TransientFaultHandling.Data]],
Sitecore.Framework.Publishing.Data</Type>
                <Options>
                  <Name>SQL Azure Backend Exponential Backoff</Name>
                  <CommandTimeout>120</CommandTimeout>
                  <RetryCount>5</RetryCount>
                  <MinBackoffSeconds>0</MinBackoffSeconds>
                  <MaxBackoffSeconds>60</MaxBackoffSeconds>
                  <DeltaBackoffSeconds>2</DeltaBackoffSeconds>
                </Options>
```

```
              </sql-backend-azure>

            <sql-api-azure>
```

```
<Type>Sitecore.Framework.Publishing.Data.AdoNet.FixedBackoffDbConnectionBehaviour`1[[Microsoft
.Practices.EnterpriseLibrary.TransientFaultHandling.SqlDatabaseTransientErrorDetectionStrategy
, Microsoft.Practices.EnterpriseLibrary.TransientFaultHandling.Data]],
Sitecore.Framework.Publishing.Data</Type>
                <Options>
                  <Name>SQL Azure API Fixed Backoff</Name>
                  <CommandTimeout>10</CommandTimeout>
                  <RetryCount>3</RetryCount>
                  <RetryIntervalSeconds>500</RetryIntervalSeconds>
                  <FirstFastRetry>true</FirstFastRetry>
                </Options>
              </sql-api-azure>
```

The two connection behaviors use the Transient Fault Handling Application Block from Microsoft to perform the retrying, and to identify a failure as being a transient failure.

For more information, see http://topaz.codeplex.com/.

- To use these connection behaviors, the ADO.NET connections that represent databases hosted on SQL Azure must be configured to use them.

  In the …\config\azure\sc.publishing.sqlazure.connections.xml.example file, you can see an example of how this configuration should be specified. It specifies the configuration to set all connections to use the SQL Azure connection behaviors and must be edited according to the deployment:

```
<Sitecore>
    <Publishing>
      <Services>
        <DefaultConnectionFactory>
          <Options>
            <Connections>
              <Links>
                <Options>
                  <Behaviours>
                    <backend>sql-backend-azure</backend>
                    <api>sql-api-azure</api>
                  </Behaviours>
                </Options>
              </Links>
              <Service>
                <Options>
                  <Behaviours>
                    <backend>sql-backend-azure</backend>
                    <api>sql-api-azure</api>
                  </Behaviours>
                </Options>
              </Service>
              <Master>
                <Options>
                  <Behaviours>
                    <backend>sql-backend-azure</backend>
                    <api>sql-api-azure</api>
                  </Behaviours>
                </Options>
              </Master>
              <Internet>
                <Options>
                  <Behaviours>
                    <backend>sql-backend-azure</backend>
                    <api>sql-api-azure</api>
                  </Behaviours>
                </Options>
              </Internet>
            </Connections>
          </Options>
        </DefaultConnectionFactory>
```

```
      </Services>
    </Publishing>
  </Sitecore>
</Settings>
```

5.12 Logging Configuration

The Microsoft Extensions Logging framework is used throughout the system to emit log messages.

For more information, see: https://github.com/aspnet/Logging.

Serilog is the default logging provider configured in the Host. This comes with a large number of sinks that can be configured for many use cases. For more information, see: https://github.com/serilog/serilog/wiki/Provided-Sinks.

Note
By default, a single file sink is configured.

The Microsoft Extensions Logging framework is based on the concept of logging levels, which are defined below in the order of significance:

- Trace
- Debug
- Information
- Warning
- Error
- Critical
- None

Each component that emits log messages in the system, by convention, does this through a logger object named with the fully qualified class name of the owning component. Therefore, there are many named loggers across the system that each emit log messages on any of the above levels.

5.12.1 Log configuration location

You can find the default logging configuration in the 'config/sitecore/sc.logging.xml' file of the publishing service installation location.

You can see an example of a logging override configuration in the 'config/development/sc.logging.development.xml' file of the publishing service installation location.

5.12.2 Configuring Logger Levels (Filters)

The level of messages that each named logger is permitted to emit can be specified in the configuration.

The `Filters` section in the example below, specifies the minimum logging level for all loggers that have a name with a matching prefix.

For example, `<Sitecore>Information</Sitecore>` specifies that only log messages at the *Information* level or above will be emitted by loggers created in the Sitecore namespace.

- To enable logging at other levels throughout the system, add additional entries, for example, `<Sitecore.Framework.Publishing.DataPromotion>Debug</Sitecore.Framework.Publishing.DataPromotion>`

If no matched filter is found the *Default* log level filter is used:

```
<Settings>
  <Sitecore>
    <Publishing>
      <Logging>
        <Filters>
          <Sitecore>Information</Sitecore>
          <Default>Warning</Default>
        </Filters>
      </Logging>
```

```
    </Publishing>
  </Sitecore>
</Settings>
```

- To customize the log levels, you override or add additional log filters. The following example adds a configuration for types in the *My.Custom.Code* namespace to log at the *Debug* level. It also changes loggers in the *Sitecore.Framework.Schedeling* namespace to log at the *Debug* level:

```
<Settings>
  <Sitecore>
    <Publishing>
      <Logging>
        <My.Custom.Code>Debug</My.Custom.Code>
        <Sitecore.Framework.Scheduling>Debug</Sitecore.Framework.Scheduling>
      </Logging>
    </Publishing>
  </Sitecore>
</Settings>
```

5.12.3 Configuring Serilog

The Serilog provider can be configured with many sinks. This configuration the default logging configuration for Serilog:

```
<Settings>
  <Sitecore>
    <Publishing>
      <Logging>
        <Filters>
          …
       </Filters>

        <Serilog>
          <WriteTo>
            <DefaultLogger>
              <Name>RollingFile</Name>
              <Args>
                <pathFormat>logs\Publishing-{Date}.log</pathFormat>
              </Args>
            </DefaultLogger>
          </WriteTo>
        </Serilog>

      </Logging>
      </Publishing>
  </Sitecore>
</Settings>
```

5.12.4 Console and File Sinks

Serilog supports many different sinks, each sink type is delivered in its own Nuget package. The Publishing Service comes with the console and file sinks included.

The default configuration above tells Serilog to put all logs produced by the service into a *logs* folder stored at the application install path, and log messages are persisted to a log file called *Publishing-<date>.log*, where *<date>* is the current date.

Logs files are treated as rolling files, where logging information is added to the file with the current date. If the log file does not exist, it is created.

You can patch in more sinks with other configuration files or replace the default one. The *config/development/sc.logging.development.xml* file adds a console logger. For example:

```
<Settings>
  <Sitecore>
    <Publishing>
      <Logging>

        <Filters>
          …
        </Filters>
```

```
        <Serilog>
          <WriteTo>
            <DevLogger>
              <Name>LiterateConsole</Name>
            </DevLogger>
          </WriteTo>
        </Serilog>

      </Logging>
    </Publishing>
  </Sitecore>
</Settings>
```

For more information on how to provide the arguments to define the parameters for these sinks, see https://github.com/serilog/serilog-settings-configuration.

For more information on the console and file logging sinks, see https://github.com/serilog/serilog-sinks-literate and https://github.com/serilog/serilog-sinks-rollingfile.

5.12.5 Other Sinks

Serilog also supports other persistence stores for log messages:
https://github.com/serilog/serilog/wiki/Provided-Sinks

To configure sinks other than Console and Rolling File for the Publishing Service:

1. Copy all the DLLs required by the sink into the Publishing Service Host directory that contains all of the Service DLLs.
2. Specify the DLL name in a *using* element in the Serilog configuration.
3. Configure the sink in the `WriteTo` section in the same way as Console and Rolling File.

Below is an example of how the Azure DocumentDB can be used to store log messages:

```
     <Serilog>
          <Using>
            <DocumentDb>Serilog.Sinks.AzureDocumentDB</DocumentDb>
          </Using>
          <WriteTo>
            <AzureLogger>
              <Name>AzureDocumentDB</Name>
              <Args>
                <endpointUri>…azure document db endpoint…</endpointUri>
                <authorizationKey>…authorization key…</authorizationKey>
                <timeToLive>3600</timeToLive>
              </Args>
            </AzureLogger>
          </WriteTo>
          <WriteTo>
            <DevLogger>
              <Name>LiterateConsole</Name>
            </DevLogger>
          </WriteTo>
        </Serilog>
```

One advantage of persisting logs to a document based database like the Azure DocumentDB, is that each log message is persisted as an object, with properties that describe the context in which the log message was emitted. Log messages can then be queried dynamically.

5.13 Troubleshooting

If you receive an error where the Internet Information Services (IIS) cannot read the application configuration, ensure you have installed all the prerequisites.

- If you receive a *502 - Bad Gateway* error when you visit your site, check the logs for details.
- After fixing any errors, restart your application pool and try again.

Chapter 6

High Availability Configuration of the Sitecore Publishing Service

This chapter describes how you can support high availability requirements by deploying multiple instances of the Publishing Service to use the same database.

- Introduction
- On premise
- Azure
- Configuration (Advanced)
- Supported Deployment Models

6.1 Introduction

When multiple Publishing Service instances are running, all of them can receive web-API calls. However, only one instance will have the job system active and therefore perform the actual publishing jobs. If the active instance fails, another instance will become active. This happens automatically because of a heartbeat protocol that is implemented via the service database.

Each service instance will request ownership of a logical lock, stored in the database, on a given schedule. Only one instance at a time can own this lock. Ownership is obtained if either there is no lock existing already, or if the current owner has not renewed the lock within a configured lifetime threshold. With the default settings, the maximum time taken to failover to a new active instance is 15 seconds, the minimum is 10 seconds.

6.1.1 Workflow

The following steps describe the workflow of when more than one Publishing Service is running against the same set of databases:

1. The service instance gets assigned a random unique name at start up, or the name can be specified explicitly in the configuration. The algorithm for generating the service name can be replaced by providing another implementation of the `IServiceInformation` interface.

   For more information about assigning a specific name to a server instance, see the section *On premise*.

2. When the service is started, the heartbeat protocol will kick off and the first instance that can access the Publishing_ActivationLock table will be set as active to enable the job system.

3. All the other instances will remain inactive, they will be able to receive API calls, but the jobs will only run on the active instance. For example, if a job is enqueued using an inactive instance, the active instance will pick it up within 10 seconds.

4. If the active instance fails, it will stop renewing the activation lock in the database. After the lock lifetime has passed, another inactive instance will be able to acquire the lock, and hence set itself as an active instance. It will then enable its job system to start processing the jobs in the queue.

5. Any job that was running when the previously active instance crashed will be automatically started by the new active instance.

6.2 On premise

In a high-availability environment, multiple instances of the Publishing Service need to be running behind a load balancer.

No special configuration is needed. However, each instance can be configured with a unique name. The configuration element is:

```
<Settings>
    <Sitecore>
      <Publishing>
        <InstanceName>${SITECORE_InstanceName}</InstanceName>
      </Publishing>
    </Sitecore>
</Settings>
```

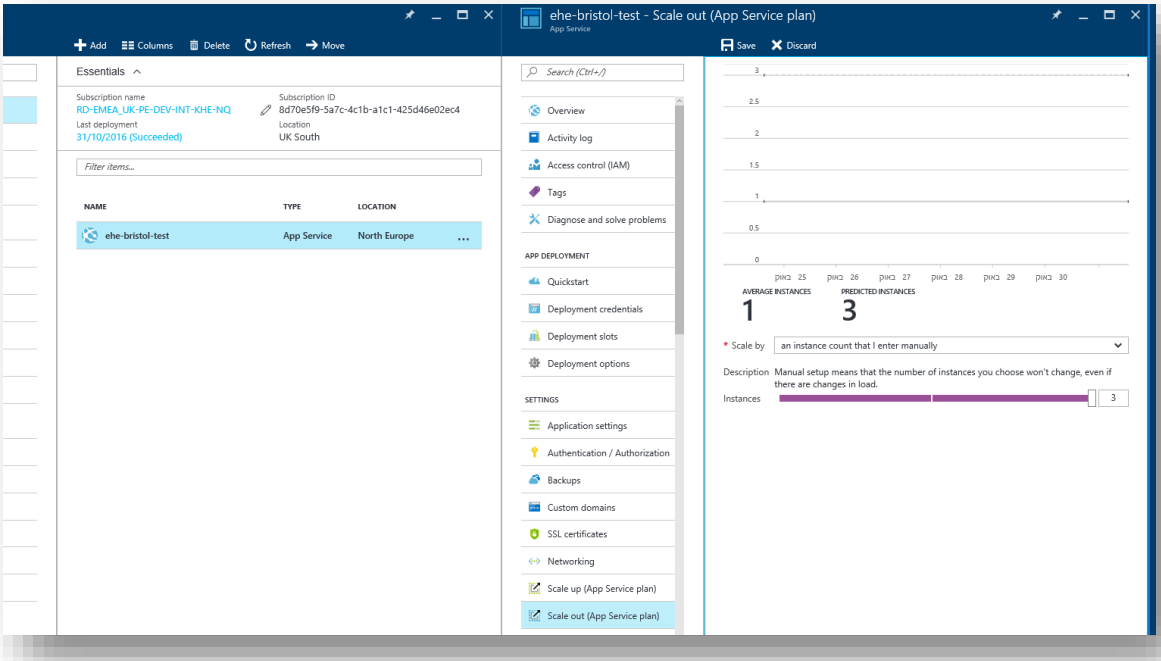
The instance name can be assigned through configuration, an environment variable, or a command line parameter. The instance name is used in logging and in the Database Publishing_ActivationLock table that shows the current active instance.

6.3 Azure

The Publishing Service can be installed as an Azure Application Service. There is no configuration needed in order to enable the high-availability functionality.

To install the Publishing Service as an Azure Application Service:

1. In the Azure portal, select a tier that allows you to use scaled-out configuration, for example, Tier B1 where you can have up to three instances.
2. Under **Settings**, click the **Scale out** option, and then drag the slider to specify the number of instances.



6.4 Configuration (Advanced)

The Publishing Service comes with defaults for the activation strategy. However, there are some parameters that can be configured if it is found that the active instance is being switched by mistake.

The following options can be configured:

- `LockAttemptIntervalInSeconds` – specify the interval in seconds that the service should use to obtain the activation lock.
- `LockRenewalIntervalInSeconds` – if the service already owns the lock, specify the interval in seconds that the service should use to renew the activation lock.
- `LockLifetimeInSeconds` – specify the interval in seconds after which the service should lose the activation lock if it hasn't renewed it, for example, in the situation where the service is inactive.

Important
Each instance must be configured with the same settings.

6.5 Supported Deployment Models

The high availability (HA) of the publishing service means that it can be used in the following configurations:

- Running on Azure as a scaled-out application service.
- Running multiple instances on multiple computers or VMs.
- Running multiple instances on the same machine. While this is not technically a high-availability setup, it can benefit testing.

Chapter 7

Publishing with the Sitecore Publishing Module

This chapter is for content authors that need to know how to publish a website or an item with the Sitecore Publishing module.

This chapter contains the following sections:

- The Sitecore Publishing Module
- Publishing an Item
- Publishing a Website
- Publish all Items
- The Sitecore Commerce Server Connect Publishing Extension Package

7.1 The Sitecore Publishing Module

In the Sitecore Publishing module, you can choose to publish the entire website or a single item:

- **Item publishing** – publishes the item you select in either the Content Editor or the Experience Editor. The item can only be published if all its ancestors have been published.

  When you publish an item, you can choose to include all its subitems and related items.

- **Site publishing** – publishes all the changes that have been made on your entire website since the last time the website was published. You can publish a site from the Content Editor or from the Sitecore Desktop.

- **Publish all items** – when you publish all items, you have the following two publishing options:

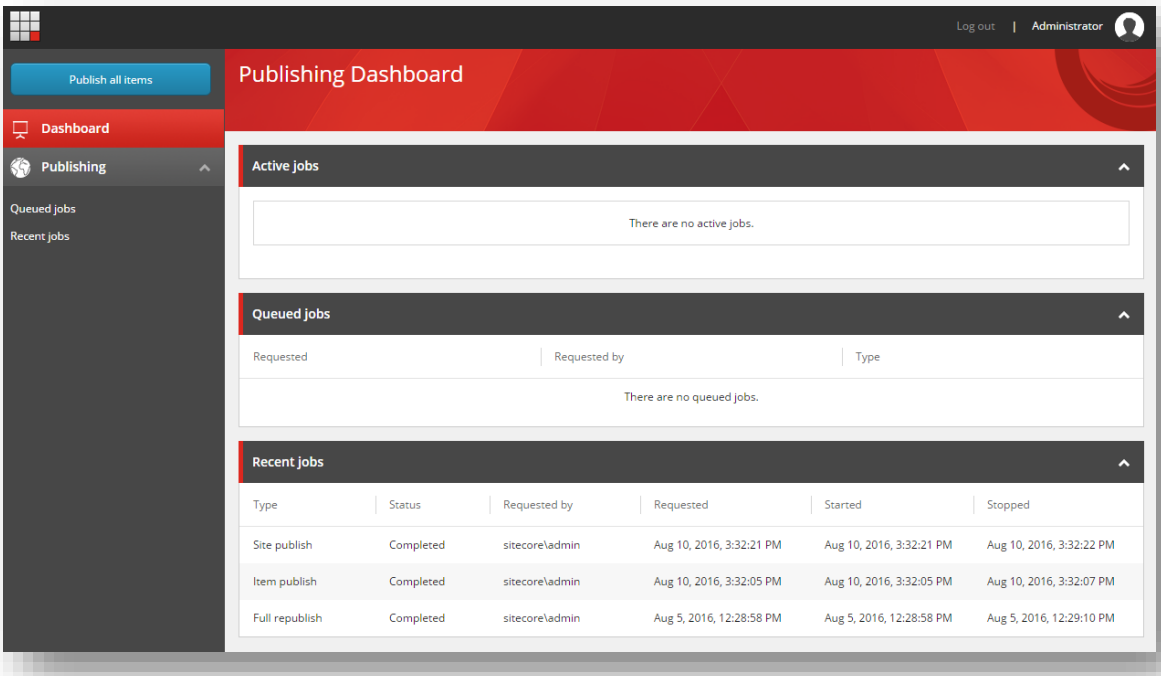
  - Publish the items that in the Master database are different from the equivalent item in the target database.
  - Publish all the items in your Sitecore installation regardless of when the items were last published. This requires a considerable amount of time and resources.

Note
Only users with sufficient access rights have the permission to perform this type of publishing.

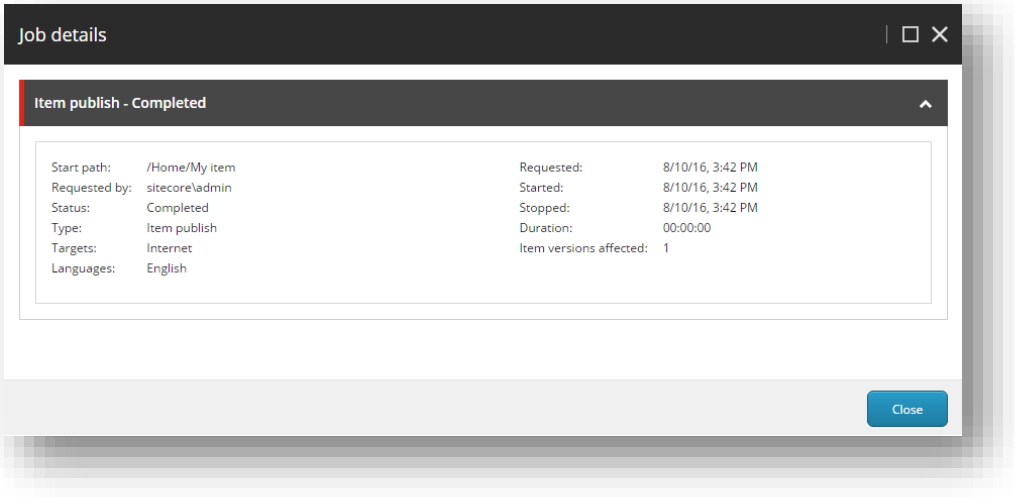
7.1.1 The Publishing Dashboard

The Publishing Dashboard gives you an overview of all the active, queued, and recent publishing jobs:

- **Active jobs** – the publishing jobs that are currently being published.
- **Queued jobs** – the publishing jobs that are waiting to be published.
- **Recent jobs** – the most recent publishing jobs that have been processed.



To see the details of a recent publishing job or a queued publishing job, click the relevant job in one of the lists.



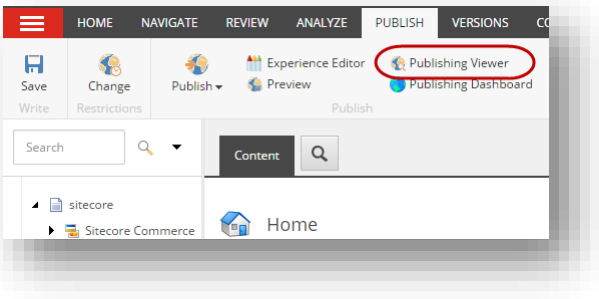
Note
There are no details to view for active publishing jobs.

7.1.2 Publishing Viewer

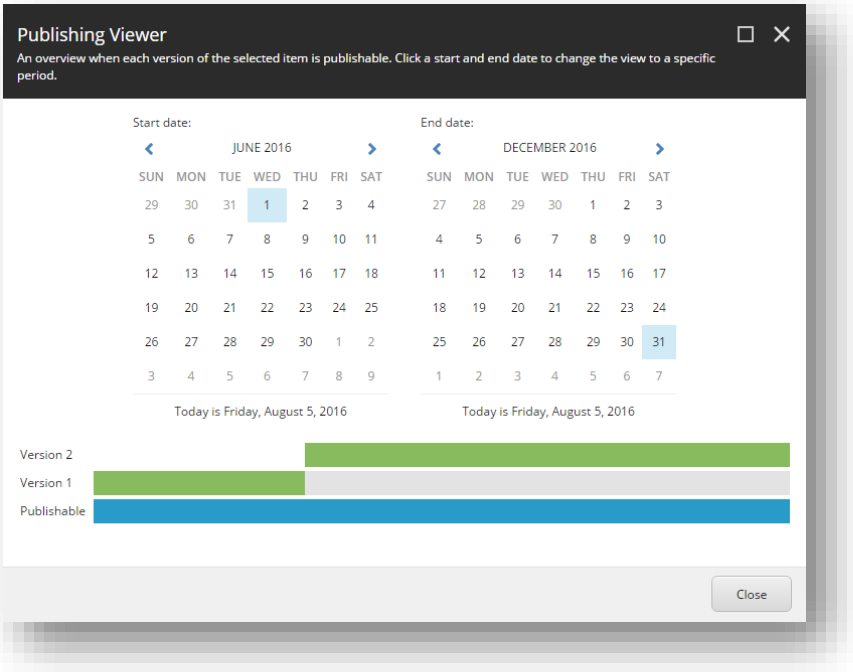
Use the Publishing Viewer to get an overview of when the various versions of an item are publishable.

To see the Publishing Viewer for an item:

1. In the Content Editor, click the relevant item.
2. On the **Publish** tab, in the **Publish** group, click **Publishing Viewer**.



3. In the **Publishing Viewer** dialog box, specify a start and end date to see if the items' versions are publishable during that period.



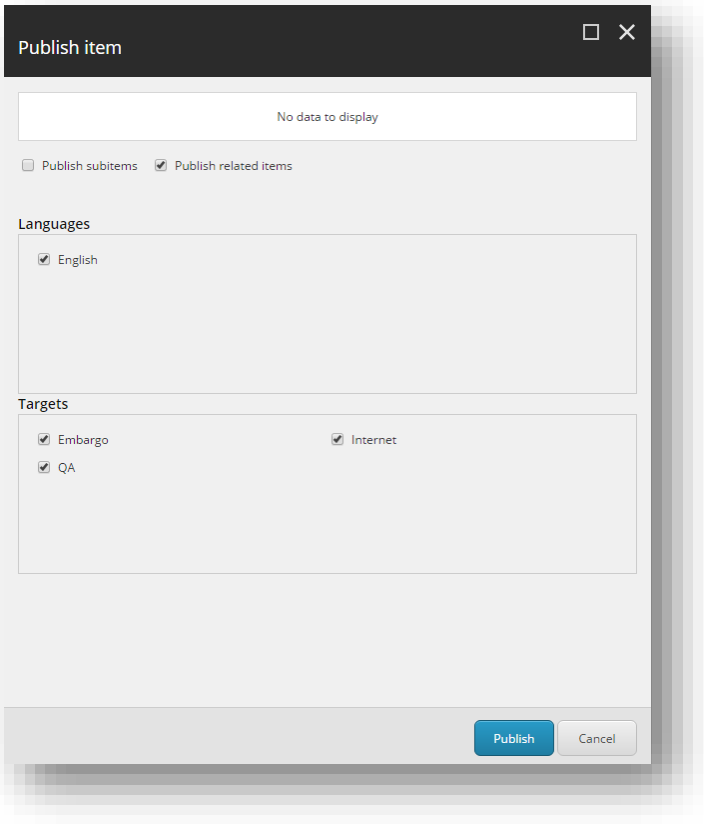
7.2 Publishing an Item

When you want to publish a single item to your website in one or more language versions, you perform an item publish from the Content Editor or the Experience Editor.

Note
To get an overview of when the different versions of an item are publishable, in the Content Editor, on the **Publish** tab, click **Publishing Viewer**.

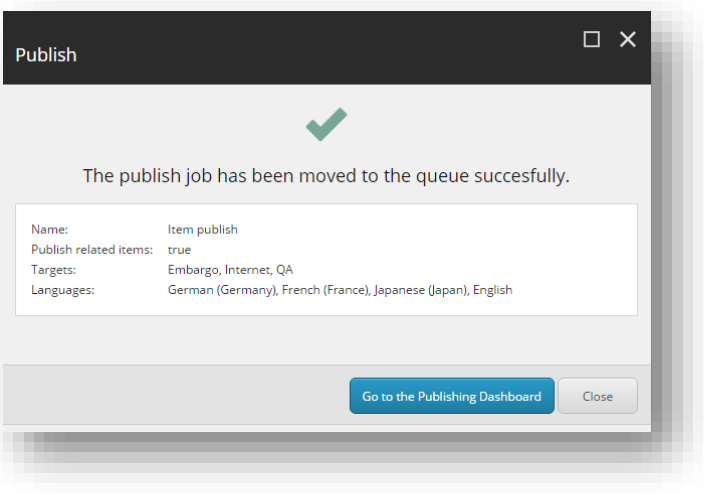
To publish a single item:

1. Open the **Publish** dialog box:
   - In the Content Editor, select the item that you want to publish. On the **Publish** tab, in the **Publish** group, click the **Publish** drop-down arrow, and then select **Publish item**.
   - In the Experience Editor, navigate to the page that you want to publish, and then on the **Home** tab, in the **Publish** group, click **Publish**.
2. In the **Publish item** dialog box, verify the item details and select:
   - **Publish subitems** to publish the current item and all its subitems.
   - **Publish related items** to publish the current item and all its related items, such as clone references, media references, and alias references.



3. Select the language versions of the item that you want to publish and the targets that you want to publish the item to.

4. To move the publishing job to the publishing queue, click **Publish**.



5. To get an overview of the active, queued, and recent publishing jobs, click **Go to the Publishing Dashboard**.

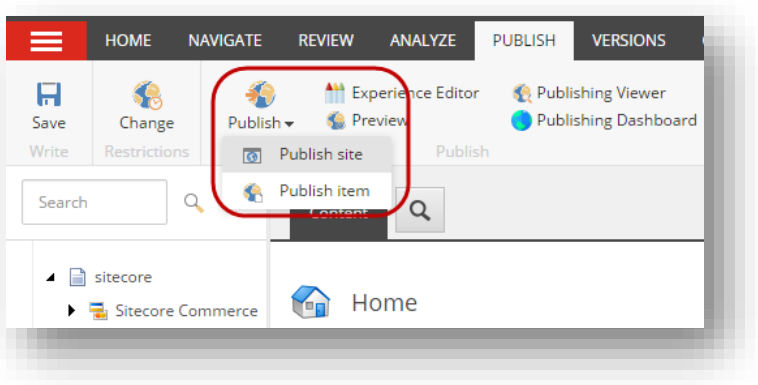
7.3 Publishing a Website

When you perform a site publish, you only publish the items that have changed since the site was last published.

Note
Users with sufficient access rights can publish all the items in your Sitecore installation at the same time regardless of when they were last published. This requires a considerable amount of time and resources. To publish all items, in the **Publishing Dashboard** click **Publish all items**.

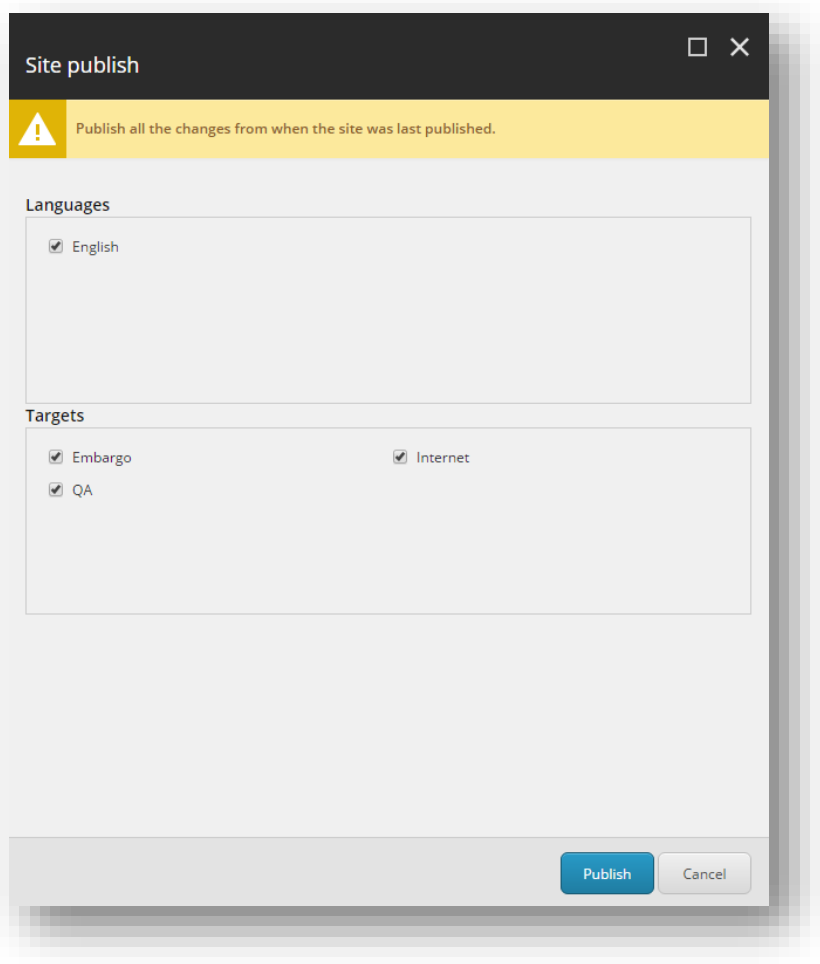
To publish the changes made to your website:

1. In the Content Editor, on the **Publish** tab, in the **Publish** group, click the **Publish** drop-down arrow, and click **Publish site**.

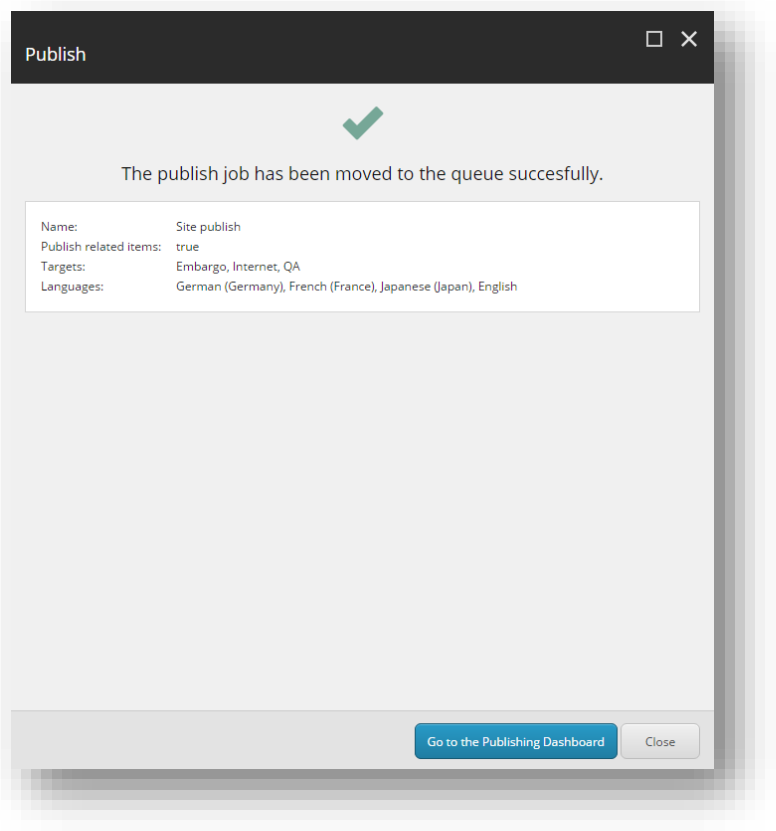


Note
You can also perform a site publish from the Sitecore Start menu.

2. In the **Publish** dialog, select the language versions that you want to publish and the targets that you want to publish the site to.



3. To move the publishing job to the publishing queue, click **Publish**.



4. To see an overview of the active, queued, and recent publishing jobs, click **Go to the Publishing Dashboard**.

7.4 Publish all Items

From the Publishing Dashboard, users that have sufficient access rights have additional two publishing options:

- Publish all the items in the Master database that are different from the equivalent item in the target database.
- Publish all the items in your Sitecore installation regardless of when the items were last published. This requires a considerable amount of time and resources.

When you publish all the items, the database and the publishing service will be subject to a much larger load than usual. Therefore, this action requires a considerable amount of time and resources.

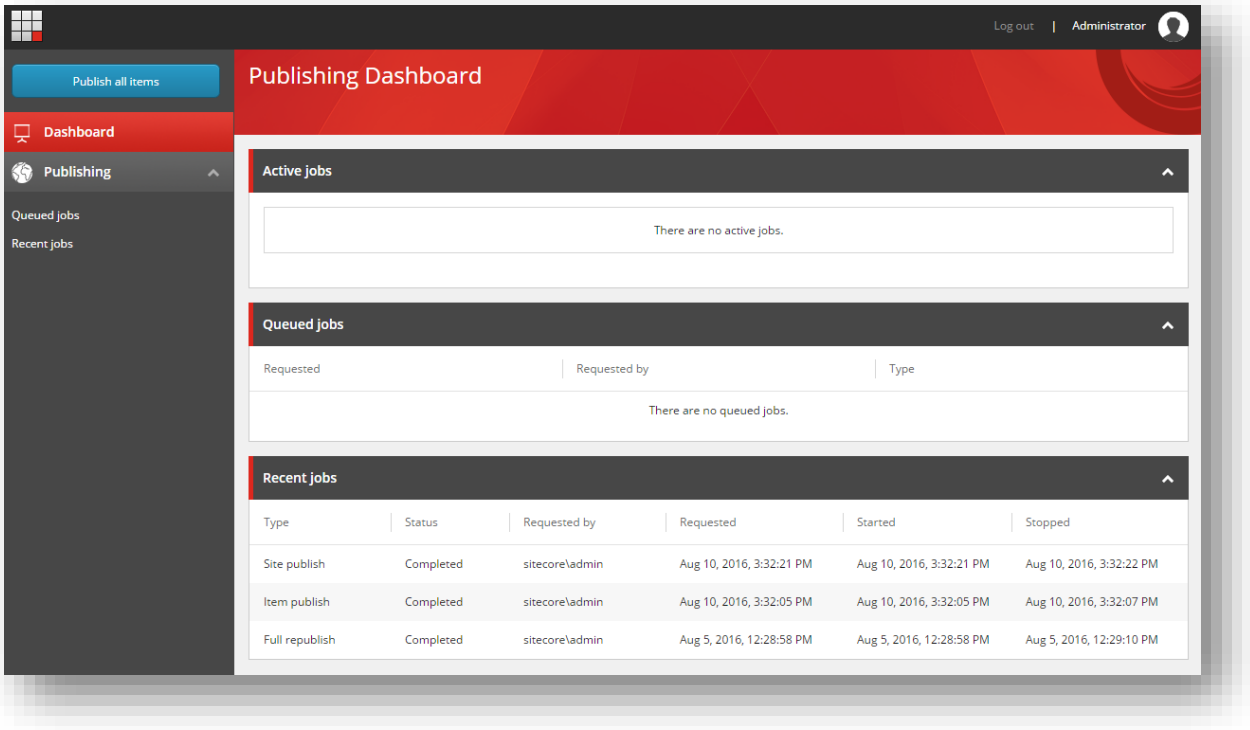
Note

To grant a specific security role permission to publish all items, you must configure the `Sitecore.Publishing.Service.Config` file. For more information, see the section Security

To publish items from the Publishing Dashboard:

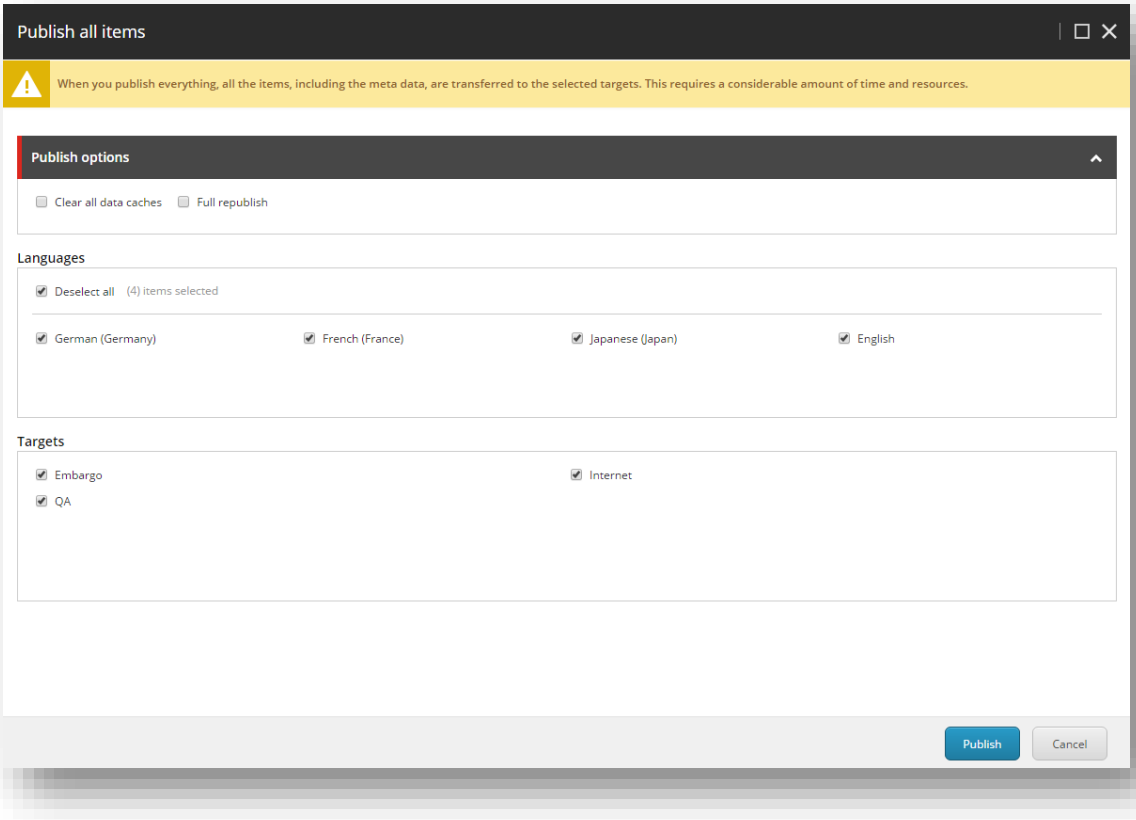
1. In the Publishing Dashboard, click **Publish all items**.

   Note
   Users that do not have the sufficient access rights cannot see the **Publish all items** button.

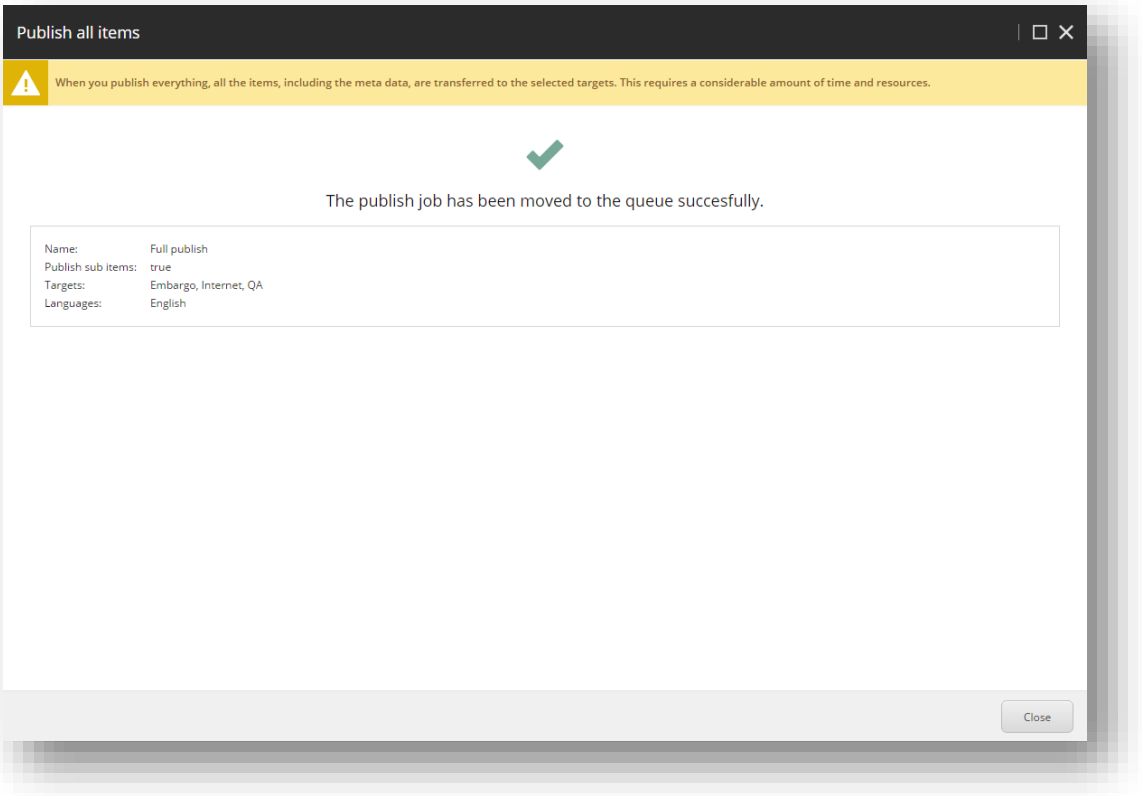


2. In the **Publish all items** dialog box:
   - Select the **Full republish** check box to publish all the items in your installation regardless of when they were last published.
   - Clear the **Full republish** check box to publish all the items in the Master database that are different from the equivalent item in the selected target database.
3. If you want to publish a large amount of items or if you want to publish to a new target, select the **Clear all data caches** check box, to clear the data level caches that contain a reference to the items that are published.

4. Select the language versions that you want to publish and the targets that you want to publish the items to.



5. To move the publishing job to the publish queue, click **Publish**.



6. To see an overview of the active, queued, and recent publishing jobs, close the **Publish all items** dialog box.

7.5 The Sitecore Commerce Server Connect Publishing Extension Package

If you have the Sitecore Commerce Server Connect Publishing extension package installed, you can start staging projects directly from the **Site publish** dialog or the **Item publish** dialog and execute staging and publishing in parallel.

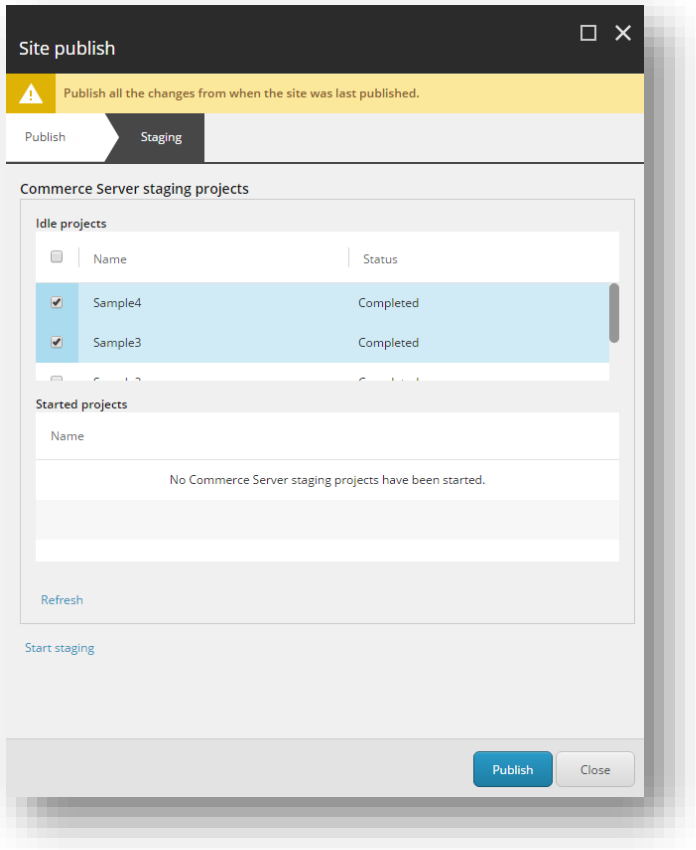
To install the extension package:

1. From http://dev.sitecore.net/ download the `Sitecore Commerce Server Connect Publishing Extensions.8.2.update` package.
2. In a browser, navigate to `http://<your site>/sitecore/admin/UpdateInstallationWizard.aspx` and then follow the steps in the installation wizard.

Note
When the installation has finished, you see a summary of the installation. The potential problems that are listed can generally be ignored.

To start a staging project when you publish:

1. In the **Site publish** or **Item publish** dialog, click the **Staging** tab.
2. In the **Idle projects** list, select the projects you would like to start and then click **Start staging**.



A notification appears to indicate that the projects have started successfully and the *Idle projects* list and the *Started projects* list are updated accordingly.

3. If you want to move the publishing job to the publishing queue, click **Publish**, otherwise click **Close**.
4. To view the status of the started staging projects, refresh the lists in the publishing dialog.

5. If you have closed the publishing dialog, just open the dialog again to see the updated status of the projects. Alternatively, you can open the Commerce Server Staging Manager.

Chapter 8

Upgrading from v1.1

One of the major differences between v2.0 and the initial v1.1 release is that there is no longer a need for separate stored procedures for the service to function. These queries have now been made part of the application itself.

If you are upgrading from version 1.1 to version 2.0, then you must remove these legacy stored procedures from any databases that have already been deployed.

- Removing Legacy Database Content

8.1 Removing Legacy Database Content

The following scripts will remove legacy stored procedures from already deployed databases when upgrading from version 1.1 to version 2.

8.1.1 Core Database

```
IF OBJECT_ID('[Publishing_Data_Delete_ItemLinks]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Data_Delete_ItemLinks]
GO

IF OBJECT_ID('[Publishing_Data_Delete_ItemsLinks]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Data_Delete_ItemsLinks]
GO

IF OBJECT_ID('[Publishing_Data_Delete_ItemVariantLinks]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Data_Delete_ItemVariantLinks]
GO

IF OBJECT_ID('[Publishing_Data_Delete_ItemVariantsLinks]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Data_Delete_ItemVariantsLinks]
GO

IF OBJECT_ID('[Publishing_Data_Get_ItemVariantInLinks]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Data_Get_ItemVariantInLinks]
GO

IF OBJECT_ID('[Publishing_Data_Get_ItemVariantOutLinks]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Data_Get_ItemVariantOutLinks]
GO

IF OBJECT_ID('[Publishing_Data_Get_ItemVariantsAllLinks]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Data_Get_ItemVariantsAllLinks]
GO

IF OBJECT_ID('[Publishing_Data_Get_ItemVariantsInLinks]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Data_Get_ItemVariantsInLinks]
GO

IF OBJECT_ID('[Publishing_Data_Get_ItemVariantsOutLinks]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Data_Get_ItemVariantsOutLinks]
GO

IF OBJECT_ID('[Publishing_Data_Set_ItemVariantsLinks]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Data_Set_ItemVariantsLinks]
GO
```

8.1.2 Master Database

```
IF OBJECT_ID('[Publishing_Data_Get_AllTemplates]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Data_Get_AllTemplates]
GO

IF OBJECT_ID('[Publishing_Data_Get_ContentUnderTest]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Data_Get_ContentUnderTest]
GO

IF OBJECT_ID('[Publishing_Data_Get_ItemNode]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Data_Get_ItemNode]
GO

IF OBJECT_ID('[Publishing_Data_Get_ItemNodeAncestors]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Data_Get_ItemNodeAncestors]
GO

IF OBJECT_ID('[Publishing_Data_Get_ItemNodeChildren]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Data_Get_ItemNodeChildren]
GO

IF OBJECT_ID('[Publishing_Data_Get_ItemNodes]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Data_Get_ItemNodes]
GO
```

```
IF OBJECT_ID('[Publishing_Data_Get_ItemVariant]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Data_Get_ItemVariant]
GO

IF OBJECT_ID('[Publishing_Data_Get_ItemVariants]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Data_Get_ItemVariants]
GO

IF OBJECT_ID('[Publishing_Data_Get_MediaChunks]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Data_Get_MediaChunks]
GO

IF OBJECT_ID('[Publishing_Data_Get_WorkflowStatesForPublish]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Data_Get_WorkflowStatesForPublish]
GO

IF OBJECT_ID('[Publishing_Data_Initialise_Params]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Data_Initialise_Params]
GO

IF OBJECT_ID('[Publishing_Delete_Job]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Delete_Job]
GO

IF OBJECT_ID('[Publishing_Delete_Manifest]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Delete_Manifest]
GO

IF OBJECT_ID('[Publishing_Delete_PublisherOperations]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Delete_PublisherOperations]
GO

IF OBJECT_ID('[Publishing_Get_Job]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Get_Job]
GO

IF OBJECT_ID('[Publishing_Get_JobsByStoppedDate]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Get_JobsByStoppedDate]
GO

IF OBJECT_ID('[Publishing_Get_JobsLastSuccessful]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Get_JobsLastSuccessful]
GO

IF OBJECT_ID('[Publishing_Get_ManifestOperationResult]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Get_ManifestOperationResult]
GO

IF OBJECT_ID('[Publishing_Get_ManifestOperationResultCount]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Get_ManifestOperationResultCount]
GO

IF OBJECT_ID('[Publishing_Get_ManifestStatus]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Get_ManifestStatus]
GO

IF OBJECT_ID('[Publishing_Get_ManifestStatusCollection]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Get_ManifestStatusCollection]
GO

IF OBJECT_ID('[Publishing_Get_ManifestStepCount]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Get_ManifestStepCount]
GO

IF OBJECT_ID('[Publishing_Get_ManifestSteps]','P') IS NOT NULL
DROP PROCEDURE[Publishing Get ManifestSteps]
GO

IF OBJECT_ID('[Publishing_Get_PublisherOperationById]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Get_PublisherOperationById]
GO

IF OBJECT_ID('[Publishing_Get_PublisherOperationsByDate]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Get_PublisherOperationsByDate]
GO
```

```
IF OBJECT_ID('[Publishing_Get_PublisherOperationsByDateAndType]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Get_PublisherOperationsByDateAndType]
GO

IF OBJECT_ID('[Publishing_Get_PublisherOperationsIdsNotEqualToType]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Get_PublisherOperationsIdsNotEqualToType]
GO
```

8.1.3 Web Database

```
IF OBJECT_ID('[Publishing_Data_Delete_Item]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Data_Delete_Item]
GO

IF OBJECT_ID('[Publishing_Data_Delete_Items]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Data_Delete_Items]
GO

IF OBJECT_ID('[Publishing_Data_Delete_ItemVariant]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Data_Delete_ItemVariant]
GO

IF OBJECT ID('[Publishing Data Delete ItemVariants]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Data_Delete_ItemVariants]
GO

IF OBJECT_ID('[Publishing_Data_Get_ItemIndexInfo]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Data_Get_ItemIndexInfo]
GO

IF OBJECT_ID('[Publishing_Data_Get_MediaExists]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Data_Get_MediaExists]
GO

IF OBJECT_ID('[Publishing_Data_Rebuild_Descendants]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Data_Rebuild_Descendants]
GO

IF OBJECT_ID('[Publishing_Data_Set_ItemVariants]','P') IS NOT NULL
DROP PROCEDURE[Publishing_Data_Set_ItemVariants]
GO
```