Sitecore Publishing Service Installation and Configuration Guide

How to install and configure the Sitecore Publishing Service

February 7, 2022
Sitecore Publishing Service 5.0.0

Table of Contents

1. Introduction

This document describes how to install and configure the Sitecore Publishing Service.

The document contains the following chapters:

- Chapter 1 - An introduction to the Sitecore Publishing Service module.
- Chapter 2 - Step-by-step instructions for installing the Sitecore Publishing Service manually or with a script.
- Chapter 3 - The various command line arguments and startup modes supported by the Sitecore Publishing Service.
- Chapter 4 - Step-by-step instructions for configuring the Sitecore Publishing Service.
- Chapter 5 - Description on how you can support high availability requirements.
- Chapter 6 - Information about the Sitecore Publishing Service API.
- Chapter 7 - Step-by-step instructions for updating the Sitecore Publishing Service.
- Chapter 8 - The Sitecore Publishing Service Support Matrix.

1.1. About the Publishing Service module

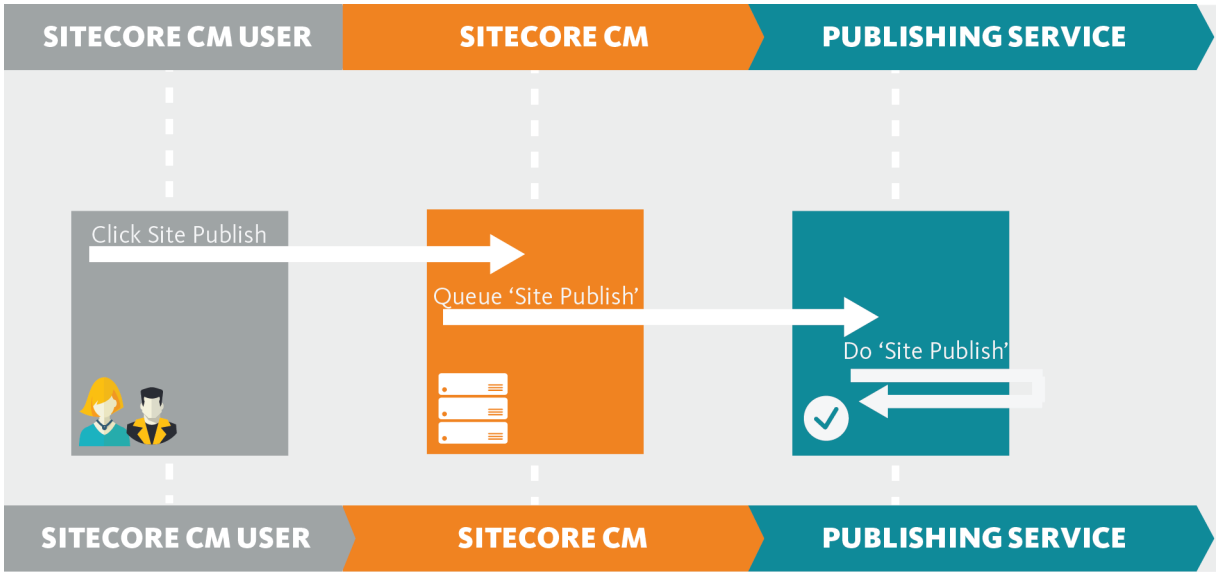
The Publishing Service module is an optional replacement for the existing Sitecore publishing methods. This module increases publishing throughput, reduces the amount of time spent publishing large volumes of items, and offers greater data consistency and reliability. The module also improves the user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

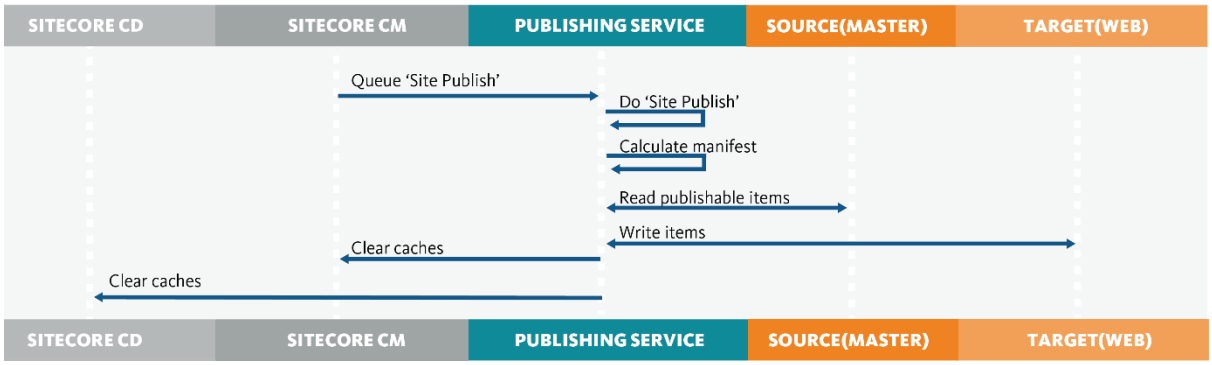
Installation involves:

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 following Publishing Service concepts relate to the different stages of the publishing work that are being handled:

- **Publishing jobs**
  Previously, when a user chose to publish something, the publishing dialog box 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 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.

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
- Scales Environment considerations

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 5.0.0 zip file.

The prerequisites for the Sitecore Publishing Service 5.0.0 release are:

- Windows Server Hosting (.NET Core)

To enable the service to run under IIS, you must install the latest version of the ASP.NET Core/.NET Core: Runtime & Hosting Bundle.

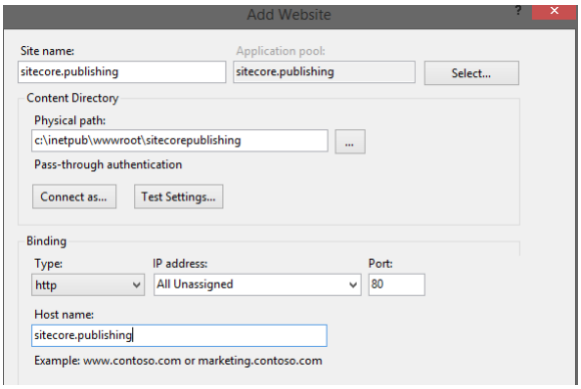
2.2. Manual installation

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

To install the Sitecore Publishing Service manually:

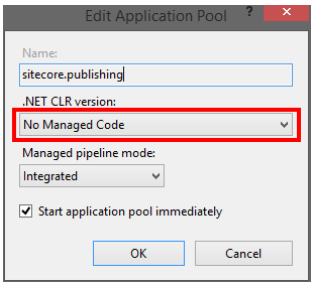
1. Download the Sitecore Publishing Service package from the Sitecore Downloads page.
2. Extract the content of the archive to a folder of your choice. For example: `C:\inetpub\wwwroot\sitecorepublishing`.
3. In IIS, create a new site pointing to the folder.
4. Start the IIS Manager and in the **Connections** 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\`).

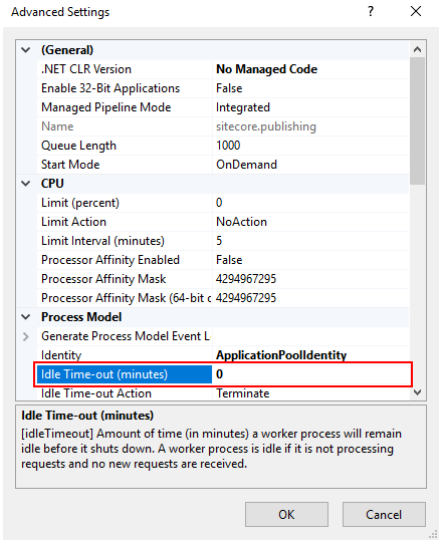
6. In the IIS Manager, right-click the application pool for the website that you created, and then click **Basic Settings**.
7. In the **Edit Application Pool** dialog, in the **.NET CLR version** field, select **>No Managed Code**.



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

8. In the IIS Manager, right-click the application pool for the website that you created, and then click **Advanced Settings**.

9. In the **Advanced Settings** dialog, in the **IdleTime-out (minutes)** field, enter *0*.



10. Configure the *core*, *master*, *web,* and *service* connection strings for the service along with any additional configuration values.
    For more information about the configuration command, see SetConnectionString Command.
11. To upgrade the database schema, run the schema upgrade command from the extracted folder:

    ```
    Sitecore.Framework.Publishing.Host.exe schema upgrade --force
    ```

    For more information about the schema upgrade command, see Upgrade.
12. To access your website, enter `http://<sitename>/api/publishing/operations/status` in your browser.
    If you receive a value of `{ "status" : 0, "statusType": "Ok" }`, 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 into 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 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 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 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 IIS Command.

2.4. 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>
```

You must also update the database schema.

To upgrade the database schema, run the following command:

```
Sitecore.Framework.Publishing.Host.exe schema upgrade --force
```

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
- ItemRevision 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 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 Sitecore configuration files.
- The ASP.NET CORE environment variables.
- The Sitecore environment variables.
- The command line.

> **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	Sitecore Environment	Command line	Type	Details
`Environment`	SITECORE_ENVIRONMENT	--environment	Single - String	Starts the service in the specified environment.

3.2.2. Custom configuration values

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

Type	Example
Configuration key	`Publishing:InstanceName`
Sitecore environment variable	`SITECORE_Publishing__InstanceName = sps_dev`
Command line	`-- --Publishing:InstanceName "sps_dev"`

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

- To 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 site name and port. The command creates two bindings based on the specified site name 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 site name
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`

Option	Template	Type	Details	Default value
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

To set or change a connection in a configuration file, use the *SetConnectionString* command.

The required arguments are:

Argument	Example	Details
Name	`Core`	Specify the name of the connection string that you want to configure.

Value	`Data Source=.\\SQLSERVER17;Initial Catalog=511108sc823_core;Integrated Security=False;User ID=***;Password=***;MultipleActiveResultSets=True;`
	
	`ConnectRetryCount=15;ConnectRetryInterval=1`
		Specify the value of the connection string.
		
		If the value does not support MARS, it will be updated.

The command options are listed below.

Option	Template	Type	Details	Default value
Help	-? | --help	Switch	Displays help information	
File Name	-f | --file	Single - String	The name of the file where changes will be persisted	sc.publishing.xml

For example:

- To set a connection string value for the Core database:

```
$ Sitecore.Framework.Publishing.Host configuration setconnectionstring core Data Source=.\
\SQLSERVER17;Initial Catalog=511108sc823_core;Integrated Security=False;User
ID=***;Password=***;MultipleActiveResultSets=True;ConnectRetryCount=15;ConnectRetryInterval=1
```

- 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.

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 the required 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.	

Option	Template	Type	Details	Default Value
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

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

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
```

3.6. ItemRevision command

With the `itemrevision` command, you can query and repair the item data stored in the Sitecore databases that relate to publishing.

3.6.1. List

Use the following options to check the format of the revision ID of each item in the Source and Target databases.

Option	Template	Type	Details
Help	-? | --help	Switch	Displays help information.
Detailed	-d | --details	Switch	Displays details of each item with an invalid revision ID.

For example:

- To list a summary of the number of invalid revision IDs in each database:
  `$ Sitecore.Framework.Publishing.Host.exe itemrevision list`
- To list each item with an invalid revision ID:
  `Sitecore.Framework.Publishing.Host.exe itemrevision list -d`

3.6.2. Fix

Use the fix command to repair the format of the revision ID of each item in the Source and Target databases.

Option	Template	Type	Details
Fix	fix	Switch	Changes each incorrect revision ID to a correct value.

For example:

- To correct each invalid revision ID for items in each database:
  `$ Sitecore.Framework.Publishing.Host.exe itemrevision fix`

3.6.3. Revision

To check and fix the format of the revision ID of each item in the source and target databases, use the following options.

Option	Template	Type	Details
Help	-? | --help	Switch	Displays help information.
Detailed	-d | --details	Switch	Displays details of each item with an invalid revision ID.
Fix	fix	Switch	Changes each incorrect revision ID to a correct value.

For example:

- To list a summary of the number of invalid revision IDs in each database:

```
$ Sitecore.Framework.Publishing.Host.exe diagnostics revision
```

- To list each item with an invalid revision ID:

```
$ Sitecore.Framework.Publishing.Host.exe diagnostics revision -d
```

- To correct each invalid revision ID for items in each database:

```
$ Sitecore.Framework.Publishing.Host.exe diagnostics revision fix
```

4. Configuring the Sitecore Publishing Service

The Sitecore Publishing Service supports custom configurations.

This chapter contains the following sections:

4.1. 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.

> **NOTE**
> 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 and resource file connections 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>
      <StageResources>
        <Type>Sitecore.Framework.Publishing.Data.ResourceItems.ResourceConnection,
Sitecore.Framework.Publishing.Data.ResourceItems</Type>
          <LifeTime>Transient</LifeTime>
           <Options>
            <Path>items</Path>
           </Options>
          </StageResources>
           </Connections>
           </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>
```

```
        <ConnectionNames>
          <stageDatabase>Stage</stageDatabase>
           <stageResources>StageResources</stageResources>
        </ConnectionNames>
           <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 configuration 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.

> **IMPORTANT**
> When you configure the `TargetStore`, it is important that the database connection name goes before the resource files connection name. Because the configuration factory can reorder elements based on their names, you must make sure that the elements in the `<ConnectionNames>` section have names that preserve the positional order.

4.2. Configuration sources

When the Publishing Service starts up it will attempt to load configuration values from various locations:

- Configuration files on disk
- Configuration contained within environment variables set on the local machine
- Configuration passed in via command line arguments

Configuration values are loaded in strict order of precedence: File, Environment Variable, Command Line.

Configuration values set by a file on disk can be overridden by the same configuration values set as environment variables which, in turn, can be overridden by the same configuration values passed in via the command line.

> **NOTE**
> If you apply any changes to any configuration files, you must restart the application for the changes to be re-loaded.

4.2.1. Configuration stored on disk

There are three locations where configuration files are loaded from:

- `<installationPath>\sitecore\Sitecore.Framework.Plugin.Publishing\config\`
  This is where the default configuration for the Publishing Service is located. These files should be considered read-only, and any changes should be made by creating your own configuration file in the user area as these files may be overridden during an upgrade.
- `<installationPath>\sitecoreruntime\_shared\config`
  You can store user configuration files in the `sitecoreruntime` folder. These normally involve overriding or patching values found in the default configuration area listed above. The configuration written to the `_shared` location will be loaded by all environments.
- `<installationPath>\sitecoreruntime\<environment>\config`
  The environment specific contains configuration for a single environment. Configuration stored here will override all previously loaded configuration.

> **IMPORTANT**
> Do not modify the default files in the default config folder. These files are automatically overwritten during the upgrade process. Instead, create your own configuration files in an environment specific folder under `sitecoreruntime`, using the configuration patching mechanisms described in this section to make the changes you require.

Runtime Environments

Configuration files on disk are collected into what is known as Environments. Environments are simple directories that contain a collection of configuration files that are loaded at runtime from within the `sitecoreruntime` directory.

When the Publishing Service starts up it will try and load files it finds in a single environment folder.

If an environment is not supplied, the Publishing Service will use `Production` as its default environment and try and load configuration from the `Production` folder (`<installationPath>\sitecoreruntime\production\config`) You can specify which environment is used by passing the environment name through the `--environment` command line parameter.

4.2.2. Configuration through environment variables

Configuration values can be set as environment variables on the local machine if required. This is useful for having machine-specific values that are stored in memory and not stored on disk (in case of security). These values will potentially overwrite the same values defined in configuration files.

4.2.3. Configuration through command line arguments

Configuration values can be passed in via the command line. These values will overwrite the same values defined on configuration files or in environment variables.

4.2.4. 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.

4.3. 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`

4.3.1. 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.

4.3.2. 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.

4.4. 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.

4.4.1. 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>
```

4.4.2. 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 Logging configuration.

4.4.3. 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 types:

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

```
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 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;
        public bool DeleteOrphanedItems { get; set; } = true;
        public bool PublishRelatedItemParents { get; set; } = false;
        public List<Guid> PublishRelatedItemDescendants { get; } = new List<Guid>();
        public IEnumerable<Guid> IgnoreChildrenOfItemIds { get; set; }
        public IEnumerable<Guid> IgnoredTemplateIds { get; set; }
        public string DefaultLanguage { get; set; } = "en";
    }
}
```

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

```
<Settings>
    <Sitecore>
        <Publishing>
            <Services>
                <IncrementalPublishHandler>
                    <Options>
                        <!--Enable this feature if content testing is required and has been
enabled in the Sitecore platform-->
                        <ContentTesting>True</ContentTesting>
                        <!--Specifies the batch size of items that are handled by the
ManifestBuilder. Increasing or decreasing this value will affect performance, depending on the
environment. -->
                        <ManifestBuilderBatchSize>5000</ManifestBuilderBatchSize>
                        <!--Enables parallel promotion, this allows the promotion of item to
multiple targets in parallel. This will speed up promotion, but consume more resources.-->
                        <ParallelPromote>False</ParallelPromote>
                        <!--Specifies the batch size of related items that are handled by the
```

```
VariantsRelatedNodesTargetProducer. Increasing or decreasing this value will affect performance,
depending on the environment. -->
                            <RelatedItemBatchSize>2000</RelatedItemBatchSize>
                        <!--Specifies the batch size of items that are handled by the
TreeNodeSourceProducer. Increasing or decreasing this value will affect performance, depending on
the environment. -->
                            <SourceTreeReaderBatchSize>2000</SourceTreeReaderBatchSize>
                        <!--Specifies the batch size of items that are handled by the
VariantsValidationTargetProducer. Increasing or decreasing this value will affect performance,
depending on the environment. -->
                            <TargetOperationsBatchSize>2000</TargetOperationsBatchSize>
                        <!--Enables transactional promotion, this performs the publish job as a
transaction. -->
                            <TransactionalPromote>True</TransactionalPromote>
                        <!--Specifies the batch size of items that are handled by the
UnpublishedNodeSourceProducer within IncrementalPublishHandler. Increasing or decreasing this
value will affect performance, depending on the environment. -->
                            <UnpublishedOperationsLoadingBatchSize>2000</
UnpublishedOperationsLoadingBatchSize>
                    </Options>
                </IncrementalPublishHandler>
            </Services>
        </Publishing>
    </Sitecore>
</Settings>
```

4.4.4. PromoterOptions

You can 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>
```

4.4.5. 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.

4.5. 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.

4.5.1. 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>Data Source=.\\SQLSERVER17;Initial Catalog=511108sc823_master;Integrated
Security=False;User
ID=***;Password=***;MultipleActiveResultSets=True;ConnectRetryCount=15;ConnectRetryInterval=1</
Master>
            <Web>Data Source=.\\SQLSERVER17;Initial Catalog=511108sc823_web;Integrated
Security=False;User
ID=***;Password=***;MultipleActiveResultSets=True;ConnectRetryCount=15;ConnectRetryInterval=1</Web>
            <Core>Data Source=.\\SQLSERVER17;Initial Catalog=511108sc823_core;Integrated
Security=False;User
ID=***;Password=***;MultipleActiveResultSets=True;ConnectRetryCount=15;ConnectRetryInterval=1</
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=.\\SQLSERVER17;Initial Catalog=511108sc823_core;Integrated Security=False;User
ID=***;Password=***;MultipleActiveResultSets=True;ConnectRetryCount=15;ConnectRetryInterval=1
```

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

4.5.2. 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

4.5.3. StoreFactory

The `StoreFactory` configuration configures stores in the application that bind 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.

4.5.4. StoreFeatureLists

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

In the following example, the features that are available to the source store are several repositories. A store feature list is linked back to a store through 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>
```

4.5.5. 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 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.Framework.Publi
shing.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.

4.6. 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>
<Commands>
<Web>
<Services>
<Add>
 <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>
</Add>
</Services>
</Web>
</Commands>
</Settings>
```

4.6.1. 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>
```

4.6.2. Schemas

The `Schemas` section names all 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>
```

4.6.3. 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.

4.7. 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.

4.7.1. 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*.

4.7.2. 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.

4.7.3. 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>
```

4.8. 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.

You must enable content availability on the Content Delivery (CD) server and on the Content Management (CM) server.

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 dependent data source has been triggered.

4.8.1. Configure content availability on the CD server and on the CM server

You must enable content availability on the Content Delivery (CD) server and on the Content Management (CM) server.

To configure content availability on the CD server and on the CM server:

1. Place the `Sitecore.Publishing.Service.Delivery.dll` in the bin directory of the server.
2. Copy the `Sitecore.Publishing.Service.ContentAvailability.config` file to the server and enable it.
3. If you use Solr for content search, enable the `Sitecore.Publishing.Service.ContentAvailability.solr.config` file.
4. 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 behavior of the Item API.

The content availability functionality adds the following fields:

- Computed fields:
  - `versionsunrisedate`
  - `versionsunsetdate`
  - `publishablefrom`
  - `publishableto`

- Virtual field:
  - `isAvailable`

Publishing service setup

To enable content availability in the Publishing Service:

1. Copy `sitecore\Sitecore.Framework.Plugin.Publishing\config\sc.publishing.contentavailability.xml.disabled` to the `sitecoreruntime\<environment>\config` directory of the Publishing Service, then enable the file by removing the `.disabled` extension.
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 see a visual representation of the date range of an item or item version and diagnose these sorts of errors.

4.9. 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 more information about 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 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.

4.9.1. 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.

4.9.2. 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 defines the command time and a retryer (by name) that are used for the connection behavior.

The `retryers` section within the `sc.publishing.services.xml` defines the configuration of the available retryers.

```
<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>
                <Retryer>NoRetryer</Retryer>
            </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>
                        <Retryer><NoRetryer/Retryer>
                    </Options>
            </sql-api-default>
        </Entries>
    </Options>
</DbConnectionBehaviours>
```

4.9.3. 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.

If you are running the Publishing Service in Azure, the `sc.publishing.sqlazure.xml` file should be applied as a configuration patch.

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 configuration 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.Configuration` Sources.

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

```
<sql-backend-azure>
    <Type>Sitecore.Framework.Publishing.Data.AdoNet.ConnectionRetryBehaviour,
Sitecore.Framework.Publishing.Data</Type>
        <Options>
            <Name>SQL Azure Backend Exponential Backoff</Name>
            <CommandTimeout>120</CommandTimeout>
            <Retryer>DefaultExponentialRetry</Retryer>
        </Options>
</sql-backend-azure>
```

```
<sql-api-azure>
    <Type>Sitecore.Framework.Publishing.Data.AdoNet.ConnectionRetryBehaviour,
Sitecore.Framework.Publishing.Data</Type>
        <Options>
            <Name>SQL Azure API Fixed Backoff</Name>
            <CommandTimeout>10</CommandTimeout>
            <Retryer>DefaultFixedIntervalRetry</Retryer>>
        </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>
```

4.10. Reporting field changes

By default, the Publishing Service reports on field changes that occur on a number of standard fields. The field changes are available in the `publishEndResultBatch` pipeline that has a processor with args of `Sitecore.PublishEndResultBatchArgs` in the `Sitecore.PublishEndResultBatchArgs` processor.

The `ReportPublishFieldsResolver` service in the `sc.publishing.services.xml` configuration file specifies the fields in which changes will be reported. You can add additional fields into the `ReportPublishFieldResolver` service, for example:

```
<ReportPublishFieldsResolver>
<options>
    <!-- Shared fields -->
    <AdditionalInvariantItemFieldsIds>
        <f1>Field Guid Here</f1>
        <f2>Field Guid Here</f2>
    </AdditionalInvariantItemFieldsIds>
  <!-- Unversioned fields -->
  <AdditionalLanguageVariantFieldsIds>
    <f1>Field Guid Here</f1>
    <f2>Field Guid Here</f2>
  </AdditionalLanguageVariantFieldsIds>
  <!-- Versioned fields -->
  <AdditionalVariantFieldsIds>
    <f1>Field Guid Here</f1>
    <f2>Field Guid Here</f2>
  </AdditionalVariantFieldsIds>
</options>
</ReportPublishFieldsResolver>
```

4.11. 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 many 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.

4.11.1. 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.

4.11.2. 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>
```

4.11.3. 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>
```

4.11.4. 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.

4.11.5. 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 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`.

Here 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.

4.12. Excluding items from automatic deletion from the target databases

When you publish all items, the Publishing Service deletes any items in the target databases that do not appear in the source database.

If there are sections of your content tree or items based on a specific template, for example, user generated content or content added from other sources, that you do not want removed automatically from the target databases, you can create a configuration file that contains the relevant item IDs of the sections and templates.

You create the configuration file using the patching mechanism.

- Under the `IgnoreChildrenOfItemIds` node, add the item IDs of sections that you want to keep in the target databases.
- Under the `IgnoredTemplateIds` node, add the item IDs of the templates that you want to keep in the target databases. In this way, the items based on the specified templates will not be removed from the target databases.
  For example:

```
<Settings>
    <Sitecore>
        <Publishing>
            <Services>
                <TreeChangesPublishHandler>
                    <Options>
                    <IgnoreChildrenOfItemIds>
                        <UserGeneratedContent>{b00accaf-ce86-408e-b606-4120356fb8cf}</
UserGeneratedContent>
                        <OtherContentSection>{a9173544-5664-4549-ad02-06d5586cb855}</
OtherContentSection>
                    </IgnoreChildrenOfItemIds>
                        <IgnoredTemplateIds>
                            <UserGeneratedContent>{a9173544-5664-4549-ad02-06d5586cb855}</
UserGeneratedContent>
                        </IgnoredTemplateIds>
                    </Options>
                </TreeChangesPublishHandler>
                <TreePublishHandler>
                    <Options>
                        <IgnoreChildrenOfItemIds>
                            <UserGeneratedContent>{b00accaf-ce86-408e-b606-4120356fb8cf}</
UserGeneratedContent>
                            <OtherContentSection>{a9173544-5664-4549-ad02-06d5586cb855}</
OtherContentSection>
                        </IgnoreChildrenOfItemIds>
                            <IgnoredTemplateIds>
                                <UserGeneratedContent>{a9173544-5664-4549-ad02-06d5586cb855}</
UserGeneratedContent>
                            </IgnoredTemplateIds>
                    </Options>
                </TreePublishHandler>
            </Services>
        </Publishing>
    </Sitecore>
</Settings>
```

4.13. Configuring the Publishing Service to use Azure Application Insights

The Publishing Service supports integration with Azure Application Insights. Application Insights provides a lot of information about your instances of the Sitecore Publishing Service.

To get the most out of the data that you are capturing with Application Insight, review the Microsoft Azure Application Insights documentation.

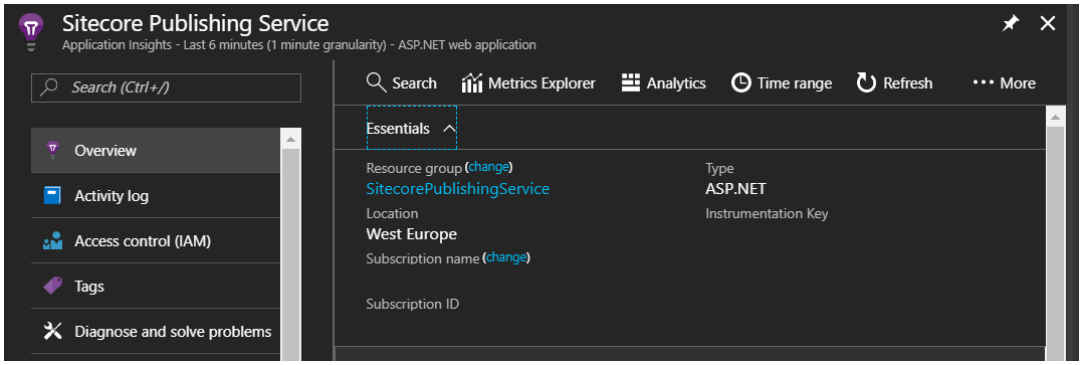
> **NOTE**
> The Publishing Service only uses the Azure Application Insights for telemetry data, such as server response times, resource utilization, number of requests, and so on. The Publishing Service logs are written to the log files that are defined in the Publishing Service configuration.

4.13.1. Prerequisites

To configure the Publishing Service to use Azure Application Insights, you must have a Microsoft Azure subscription and at least one Application Insights instrumentation key from an active Application Insights service.

- To create an Application Insights service, refer to the Microsoft Azure documentation.

When you have a running Application Insights service, you find the instrumentation key in the **Essentials** panel.



4.13.2. Configure the Publishing Service to use Application Insights

To configure the Publishing Service to use Application Insights, you edit the `appsettings.json` file in the root directory of the Publishing Service.

For example, if you have installed the Publishing Service to `C:\inetpub\wwwroot\SitecorePublishingService`, then the path to your `appsettings.json` file is `C:\inetpub\wwwroot\SitecorePublishingService\appsettings.json`.

The file must contain the following code, where you populate the value of the `InstrumentationKey` property with the one from your own Application Insights service.

```
{
    "ApplicationInsights": {
        "InstrumentationKey": ""
```

```
    }
}
```

You can also supply the configuration files for different environments using different instrumentation keys. For example, if you want to use a different instrumentation key in a development environment, then you can create an `appsettings.development.json` file in the root directory of the Publishing Service and use a different value for the `InstrumentationKey` property.

> **NOTE**
> The `appsettings.<environment>.json` file that you use matches the environment that is configured for the Publishing Service. For more information, see Host Configuration Options.

4.13.3. Adding Serilog.Sinks.ApplicationInsights

The Sitecore Publishing Service uses the Serilog logging framework to generate log information at runtime. Microsoft Application Insights can also be configured to log information on the Azure platform. Serilog also provides the ability to communicate with Microsoft Application Insights. By combining the two, the Publishing Service can publish structured logs to Application Insights, enabling management of logs within the Azure platform.

When Application Insights is configured, you must configure Serilog to send logging formation to Application Insights.

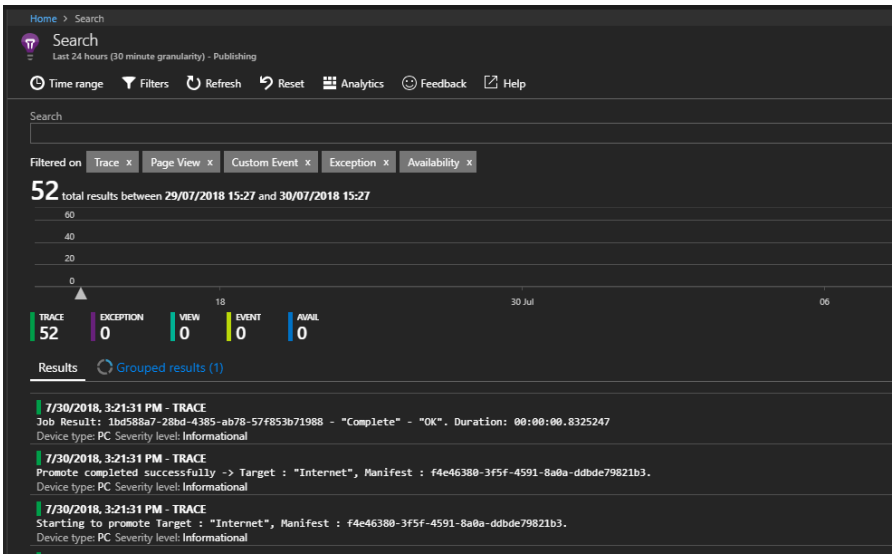
To configure Serilog to send logging formation to Application Insights:

1. Download *Serilog.Sinks.ApplicationInsights* version 2.6.0 from *nuget.org*.
2. Open the nuget package and copy the `\lib \net45\Serilog.Sinks.ApplicationInsights.dll` file into the root folder of the Publishing Service that you have deployed.
3. Create a configuration file called `sc.customlogging.xml` and add the following:

```
<Settings>
    <Sitecore>
        <Publishing>
            <Logging>
                <Serilog>
                    <WriteTo>
                        <AppInsightsLogger>
                        <Name>ApplicationInsightsTraces</Name>
                        <Args>
                            <instrumentationKey>c68d16ec-4df4-4d7a-85a3-632687961c82</
instrumentationKey>
                        </Args>
                    </AppInsightsLogger>
                    </WriteTo>
                </Serilog>
         </Logging>
        </Publishing>
    </Sitecore>
</Settings>
```

4. Update the `<instrumentationKey>` to your own value.
5. Add this configuration file to the `Publishing Service\config\global` folder.

Perform a few publishes and within a few minutes you'll be able to see the trace entries that are logged.



4.14. 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.

5. 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

5.1. 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.

5.1.1. 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 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.

5.2. 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>
```

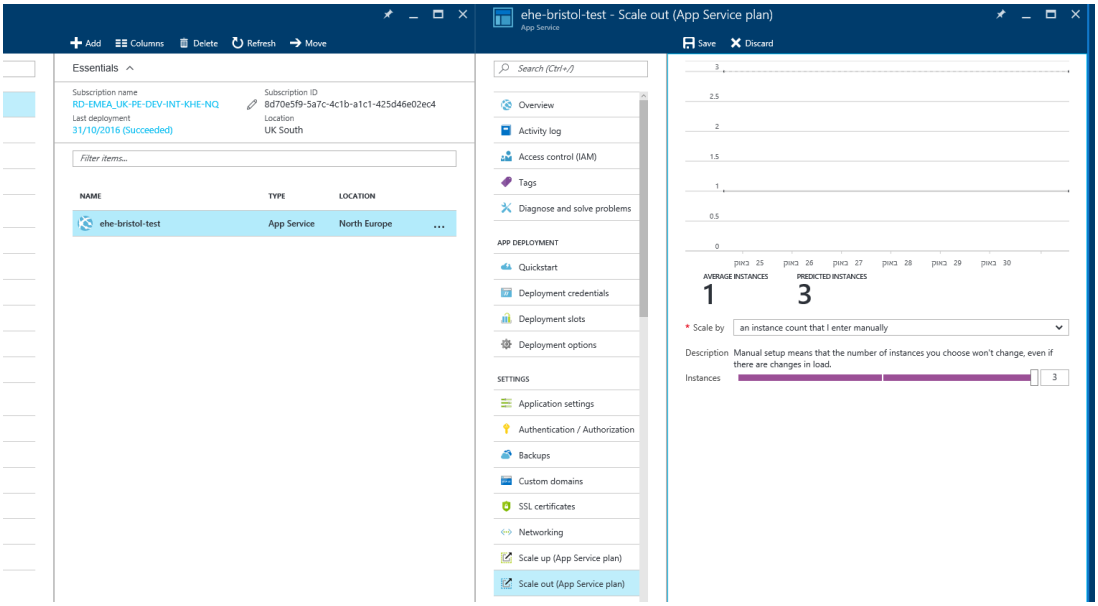
The instance name can be assigned through configuration, an environment variable, or a command line parameter. The instance name is used in logging and in the `Database Publishing_ActivationLock` table that shows the current active instance.

5.3. 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.



5.4. 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.

You can configure the following options:

- `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.

5.5. 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.

6. Publishing Service API

Every interaction between the Sitecore Publishing Service module in Sitecore and the Sitecore Publishing Service is performed with web requests to the Publishing Service web API.

Consumers that want to interact directly with the Publishing Service can also use the Publishing Service web API.

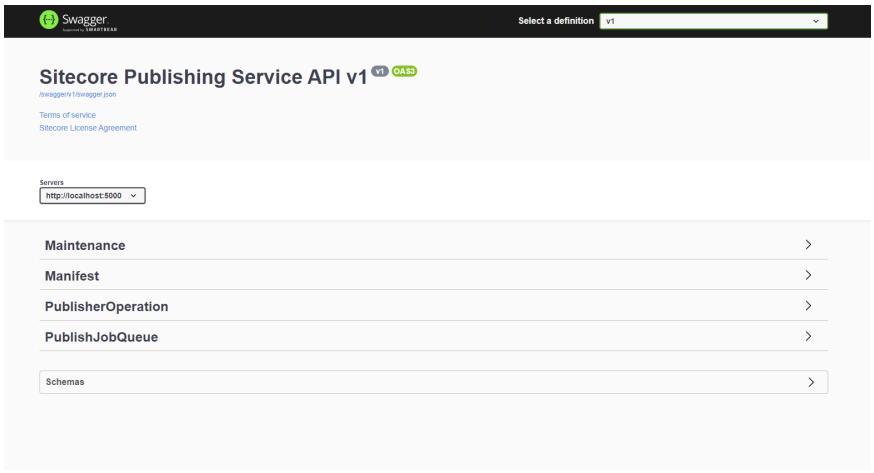
This chapter contains the following section:

- API documentation

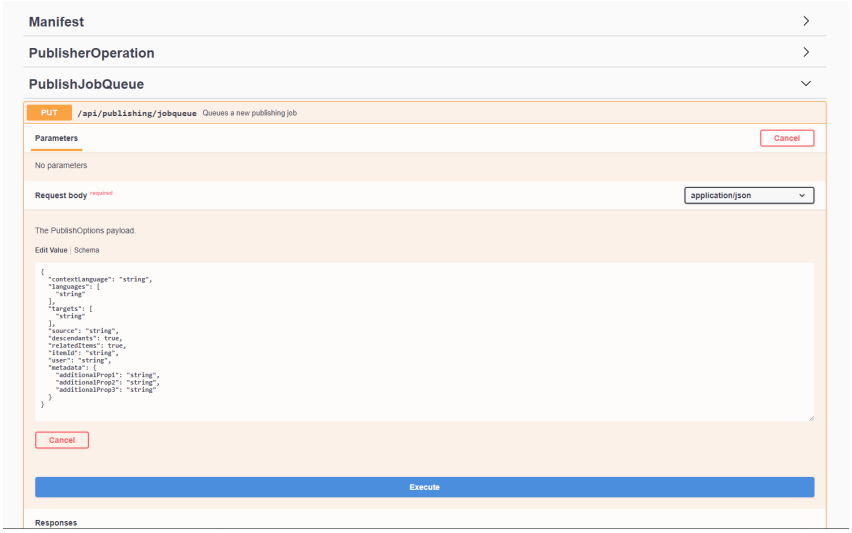
6.1. API documentation

Swagger libraries of all the API endpoints are available for the Publishing Service.

> **IMPORTANT**
> If the Publishing Service is running in a *development* environment, you can access the Swagger UI at http://localhost:5000/swagger/index.html



The main Swagger dashboard describes all the endpoints that the Publishing Service exposes. You can explore the different API versions and the endpoints from the dashboard as well as invoke and test the API methods.



To generate Swagger specification files (`swagger.json`) for the Publishing Service APIs, use the `apigen` command:

```
$ Sitecore.Framework.Publishing.Host apigen [target-path]
```

In this example, [target-path] is the folder where the `apigen` command saves the specification files.

7. Upgrading to Version 5.0.0

You can upgrade to Sitecore Publishing Service 5.0.0 from versions 3.1.x or later.

This chapter contains the following section:

7.1. Upgrade the Publishing Service

Before you upgrade to a new version of the Publishing Service, make sure you create a backup of the existing Publishing Service directory.

To upgrade the Publishing Service:

1. Ensure that you meet the prerequisites for the Publishing Service.
2. For the installed Publishing Service instance, stop the IIS application pool.
3. Extract the content of the `Sitecore Publishing Service 5.0.0.zip` file to the existing Publishing Service directory and choose to replace all files.
4. Use the `Sitecore.Framework.Publishing.Host` configuration and the `setconnectionstring` command to reconfigure your connection string.
   If your connection strings are not changed, you must use a schema command to reconfigure your schema.
5. Start the IIS application pool again.

8. Publishing Service support matrix

Starting with Sitecore XP 9.2.0, the Sitecore Publishing Service Module uses the same version number as the Sitecore Platform and not the Publishing Service. The Publishing Service still uses its own version numbering.

We recommend updating to the latest version of the Publishing Service and Publishing Service Module at the earliest opportunity. The support matrix details which versions of the Sitecore Platform are supported:

Sitecore Platform	Publishing Service Module	Publishing Service
8.2 Initial Release – 8.2 Update 1	1.1 Initial Release	1.1 Initial Release
8.2 Update-2	2.0 Initial Release	2.0 Initial Release
8.2 Update-2 – 8.2 Update-3	2.0 Update-1	2.0 Update-1
8.2 Update-2 – 8.2 Update-5	2.1 Initial Release	2.1 Initial Release
8.2 Update-2 – 8.2 Update-7	2.2 Initial Release	2.2 Initial Release
8.2 Update-2 – 8.2 Update-7	2.2 Update-1	2.2 Update-1
9.0 Initial Release	3.0 Initial Release	3.0 Initial Release
9.0 Initial Release – 9.0 Update-1	3.1 Initial Release	3.1 Initial Release
9.0 Initial Release – 9.0 Update-2	3.1 Update-1	3.1 Update-1
9.0 Initial Release – 9.0 Update-2	3.1 Update-2	3.1 Update-2
9.0 Initial Release – 9.0 Update-2	3.1 Update-3	3.1 Update-3
9.1 Initial Release	9.1 Initial Release	4.1.0 – 4.3.0
9.1 Update-1	9.1 Update-1	4.1.0 – 4.3.0
9.2.0	9.2.0	4.1.0 – 4.3.0
9.3.0	9.3.0	4.1.0 – 4.3.0
10.0.0	10.0.0	4.1.0 – 4.3.0
10.1.0	10.1.0	5.0.0