Sitecore Installation Framework Configuration Guide

Sitecore XP 9.1.1

Step by step guide to configuring Sitecore 9.1.1 with the Sitecore Install Framework

sitecore®
Own the experience™

Table of Contents

Chapter 1

Introduction

This document describes the Sitecore Installation Framework for Sitecore 9.1.1.

This document contains the following chapters:

- **Chapter 1 – Introduction**
  An introduction to the Sitecore Installation Framework.
- **Chapter 2 – Install the Module**
  Contains information about the steps necessary to install the Sitecore Installation Framework module.
- **Chapter 3 – Customize the Sitecore Installation Framework**
  Contains information about extension points that allow you to customize the Framework for your specific installation needs.
- **Chapter 4 – Further Guidance**
  Contains information about the built-in help system from Microsoft.

1.1 Getting Started

The Sitecore Installation Framework is a Microsoft® Powershell module that supports local and remote installations of Sitecore, and it is fully extensible. You can install the entire Sitecore solution (XP), or the CMS-only mode (XM) solution.

This Sitecore Installation Framework Configuration Guide describes how to configure and customize the installation process of the Sitecore Experience Platform (XP) to suit your own needs.

1.1.1 How to Use This Guide

This guide describes how to configure your Sitecore XP installation with the Sitecore Installation Framework, as well as cmdlets and extensibility points. There is also a chapter on further help and guidance.

This guide is intended to be a supplement to the Sitecore Installation Guide, to help you customize your installation. Refer to the Installation Guide for information about system requirements and other prerequisites, the general installation process, and the additional configuration tasks that are necessary after installing Sitecore XP.

You can download the Sitecore XP Installation Guide from the Sitecore Downloads page – https://dev.sitecore.net.

Chapter 2

Install the Module

This chapter contains information about installing, updating, validating, and importing the Sitecore Installation Framework module.

This chapter contains the following section:

- Install the Sitecore Installation Framework Module

2.1 Install the Sitecore Installation Framework Module

You can install the Sitecore Installation Framework module directly through Microsoft PowerShell®, or you can perform a manual installation by downloading Sitecore Installation Framework module as a .ZIP package.

2.1.1 Installation with Microsoft PowerShell®

The Sitecore Installation Framework is made available through the Sitecore Gallery. The Sitecore Gallery is a public MyGet feed where you can download PowerShell modules created by Sitecore.

To install Sitecore Install Framework with PowerShell:

1. To add the repository, in Windows, launch Microsoft PowerShell® as an administrator and run the following cmdlet:

```
Register-PSRepository -Name SitecoreGallery -SourceLocation
https://sitecore.myget.org/F/sc-powershell/api/v2
```

2. To install the PowerShell module, run the following cmdlet:

```
Install-Module SitecoreInstallFramework
```

3. When prompted to install the module, press Y, and then ENTER.

Update the Sitecore Installation Framework Module

As new features and bug fixes are periodically released, it is recommended that you update the Sitecore Installation Framework.

Note
This procedure is optional.

- To update the Sitecore Installation Framework module, run the following cmdlet:

```
Update-Module SitecoreInstallFramework
```

2.1.2 Manual Installation

The Sitecore Installation Framework is also provided as a `.ZIP` package. You can download the Sitecore Installation Framework from the Sitecore Downloads page – https://dev.sitecore.net.

When you download the package, the `.ZIP` package may be marked as *blocked* by Microsoft Windows. To continue the installation of the Sitecore Installation Framework, you must first unblock the `.ZIP` package.

Unblock a .ZIP package

To unblock a `.ZIP` package:

1. In Windows Explorer, navigate to the folder where you downloaded the packages, and right-click the relevant `.ZIP` file.
2. Click **Properties**.
3. In the **Properties** dialog box, on the **General** tab, click **Unblock**.
4. Click **OK**.

Extract the Sitecore Installation Framework

The installation path that you use depends on where you want to install the Sitecore Installation Framework. You can install it for all users (global path), for a specific user, or to a custom location.

Usage	Path
All users	`C:\Program Files\WindowsPowerShell\Modules`
Specific user	`C:\Users\<user>\Documents\WindowsPowerShell\Modules`
Custom location	Any path

For example, if you want to make the Sitecore Installation Framework available to all users, extract the `Sitecore Install Framework.ZIP` package to the following folder:

```
C:\Program Files\WindowsPowerShell\Modules\SitecoreInstallFramework
```

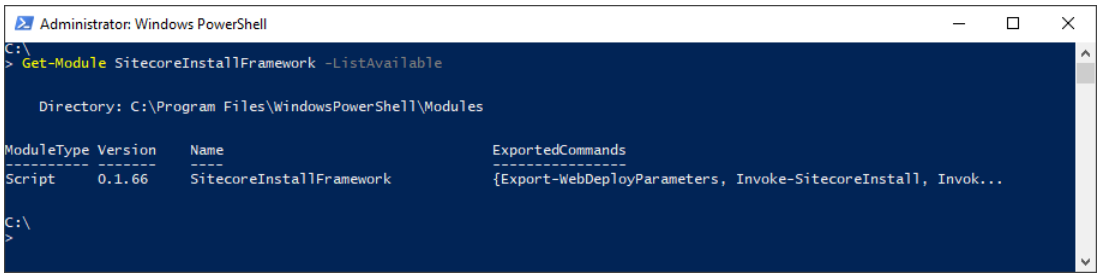
2.1.3 Validate the Installation

After you install the Sitecore Installation Framework, you can validate the installation to confirm that it is available for use. This procedure is optional.

Note
This validation only works if you have installed the Sitecore Installation Framework for *All users* (global).

- To validate the installation, run the following cmdlet:

```
Get-Module SitecoreInstallFramework -ListAvailable
```



2.1.4 Import the Sitecore Installation Framework into a Powershell session

If you added the Sitecore Installation Framework to either an *All users* or *Specific user* path, you do not have to import it, as this is done automatically, and you can immediately use it in a session by running any cmdlet available in the PowerShell module.

However, if you have installed the Sitecore Installation Framework to a custom location, run the following cmdlet.

```
Import-Module C:\<CustomLocation>\SitecoreInstallFramework
```

2.2 Multiple Versions of SIF

If you want to install a 9.0.x version of Sitecore XP on the same computer as a 9.1 installation, you must also have SIF 1.2.1 installed. PowerShell uses the latest available version of a module in a session by default.

To install a specific version of SIF run the following command:

```
Install-Module -Name SitecoreInstallFramework -RequiredVersion x.x.x
```

Enter the appropriate value in the RequiredVersion parameter.

The following table lists the versions of SIF that are compatible with Sitecore XP 9.X:

Sitecore XP Version	SIF Version
9.0.x	1.2.1
9.1	2.0.0
9.1.1	2.1.0

2.2.1 Running a specific version of SIF

To run a specific version of SIF, launch a new Powershell session and run the following command:

```
Import-Module -Name SitecoreInstallFramework -Force -RequiredVersion x.x.x
```

You will use the specified version for the remainder of the session.

The next time you start a PowerShell session it will automatically use the latest available version.

Chapter 3

Customize the Sitecore Installation Framework

The Sitecore Installation Framework enables you to customize the installation process by using a standard configuration design that you can extend by using custom PowerShell functions. The framework defines a configuration format that supports tasks, parameters, config functions, and variables. For example, you can configure a computer with one or more Sitecore instance, add services, or add custom applications.

All Sitecore Installation Framework configurations are written as `.JSON` files.

This chapter contains the following sections:

- Create and Customize Configurations
- Create Tasks
- Create Config Functions

3.1 Create and Customize Configurations

You can use tasks, parameters, config functions, and variables to customize your installation process. Custom tasks and config functions can also be packaged as a module, and be included in configurations.

You can base your customization on one of the configurations provided by Sitecore, or you can create your own configuration. This section describes the different components that you can use in a Sitecore Installation Framework configuration.

3.1.1 Tasks

Tasks are actions that are conducted in sequence when the `Install-SitecoreConfiguration` cmdlet is called. A task is implemented as a PowerShell cmdlet.

Each task is identified by a unique name and must contain a `Type` property. A task can have parameters or a collection of parameters passed to it. Tasks map directly to PowerShell functions that are registered with `Register-SitecoreInstallExtension -Type` Task.

The following example shows a task of type `Copy` that you can use in a configuration to copy files from one location to another:

```
{
    "Tasks": {
      "CopyFiles" : {
        "Description": "Copies files to the specified location",
        "Type": "Copy",
        "Params": {
          "Source": "c:\files",
          "Destination": "c:\newfiles"
        }
      }
    }
}
```

Skipping Tasks

A task can include a `Skip` property that refers to a parameter, variable, or config function. If the return value is `true`, the task is not executed.

```
{
    "Parameters : {
        "Param1" : {
            "Type" : "String"
            "DefaultValue" : ""
        }
    },
    "Tasks" :{
        "Task1" :{
            "Type" : "Copy
            "Params" :{
                "Source" : "C:\\Source",
                "Destination : "C:\\Destination"
            }
            "Skip" : "[not(Parameter('Param1'))]"
        }
    }
}
```

Note
To ensure that your configuration is valid, each task must have a unique name. This means that the task can be directly executed, and the task can be identified in a log. It also enables the same type of task to be used multiple times in a configuration.

In this configuration, if Param1 is set to any value, `Task1` is skipped.

Requires (Prerequisites)

A task may include a `Requires` block that allows prerequisite checks to be performed prior to execution of the task. The prerequisite checks are performed after the `Skip` section. Prerequisites should take the form of Config Functions which return a boolean. If the prerequisite check returns false then the Sitecore Install Framework will attempt to enter a nested shell within the current host. You will then have an opportunity to correct the problem, either from the prompt provided or externally. Once you have corrected the problem enter `exit` to return to the Sitecore Install Framework. The prerequisite checks will be performed again. Alternatively the prerequisite can be skipped by entering the command `SkipRequire`. If the current host doesn't support a nested shell then the requirements will be skipped.

```
{
    "Variables": {
        "Requires.Success" : [TestPath(Path:'C:\\Windows')]
    },
    "Tasks": {
        "Task1":{
            "Type": "WriteOutput",
            "Params" :{
                "InputObject":"Simple Task, depends on a variable."
            },
            "Requires": "[variable('Requires.Success')]"
        }
    },
    "Settings" : {
        "AutoRegisterExtensions" : True
    }
}
```

In this configuration, `Task1` requires the output of the `Requires.Success` variable to check the presence of `C:\Windows` to be true

3.1.2 Parameters

Parameters let users change values inside a configuration at runtime. Parameters must declare a type. They can also declare a default value and a description.

When you add a parameter to a configuration, it can be passed into a task using the `parameter` config function. The task then points to the value provided in the configuration or the value passed when `Install-SitecoreConfiguration` is called.

For example, to pass the `Source` and `Destination` parameters to the `CopyFiles` task:

```
{
    "Parameters": {
      "Source": { "Type": "string", "Description": "The source of files" },
      "Destination": { "Type": "string", "DefaultValue": "c:\newfiles" }
    },
    "Tasks": {
      "CopyFiles" : {
        "Type": "Copy",
        "Params": {
          "Source": "[parameter('Source')]",
          "Destination": "[parameter('Destination')]"
        }
      }
    }
}
```

On the one hand, the `Source` parameter does not contain a *DefaultValue* property, so it is required when `Install-SitecoreConfiguration` is called.

However, on the other hand, the `Destination` parameter does have a default value. If the user does not provide a value, the `DefaultValue` from the configuration is used.

The values at runtime are then passed to the `CopyFiles` task using the `parameter` config function.

To pass the values at the command line:

- You must use the name of the parameter with the standard PowerShell parameter syntax.

For example: `Install-SitecoreConfiguration -Path .\configuration.json -Source c:\sourcefiles`

Parameters Validation

Parameter values can be validated before any tasks are started. Validation logic can be specified for each parameters using config functions:

```
{
    "Parameters": {
        "Source": {
            "Type": "string",
            "DefaultValue": "c:\\myfiles",
            "Validate": "[validatelength(4, 260, $ )]"
        },
    }
}
```

In this example the parameter source is validated using the `validatelength` config function, which ensures the value is between 4 and 260 characters long.

The `Validate` property accepts any config function that returns `bool`.

This is a list of the predefined functions that you can use for validation:

- `ValidateCount` - Validates that the parameter array length is within the specified range.
- `ValidateLength` - Validates that the parameter length falls within the range.
- `ValidateNotNull` - Verifies that the argument is not null.
- `ValidateNotNullOrEmpty` - Validates that the argument is not null and is not an empty string.
- `ValidatePattern` - Validates that the parameter matches the RegexPattern.
- `ValidateRange` - Validates that number parameter falls within the range specified by `Min` and `Max`.
- `ValidateSet` - Validates that the parameter is present in a specified set.

When `Install-SitecoreConfiguration` is called, all the parameters with validation logic are checked. If any validation fails, the installation command is aborted. You can bypass validation by calling `Install-SitecoreConfiguration` with the `-SkipValidation` option.

3.1.3 Config Functions

Config functions allow elements of the configuration to be dynamic, and allow you to calculate values, invoke functions, and pass these values to tasks so that a configuration can be flexible.

A config function is written as a string enclosed in square brackets [ ]. It is identified by a name and can receive function parameters. For example: `[functionName(param1,param2,param3)]`.

Each function maps directly to a PowerShell function that is registered with the following cmdlet:

```
Register-SitecoreInstallExtension -Type ConfigFunction
```

Named Parameters

You can reference the parameters of a config function directly to ensure that the correct values are passed to the expected parameter.

Named parameters are a comma separated list denoted by following the parameter name with a colon:

```
{
    "Variables": {
        "Variable1": "[GetFunction(Number:1234,String:'Text',Switch:True)]"
    }
}
```

In this example `Variable1` calls the `GetFunction` function and passes it the named parameter `Number` with the value *1234*, the named parameter `string` with the value `'Text'` and the switch parameter `Switch` with the value `true`. This is the equivalent of typing `Get-Function -Number:1234 -String:"Text" -Switch:$True` in the command line

Switch Parameters, if declared, must be set within the configuration to `True` or `False`.

Nested functions are also permitted and may refer to parameters, variables, or other config functions. For example:
`[FunctionName1(functionName2(Switch1:'value1',Switch2:'value2'),Switch3:'value3')]`

3.1.4 Variables

Variables are values that are calculated within the configuration itself. Unlike parameters, variables can use config functions to determine their value. Variables cannot be overridden at runtime; however, when they are being calculated, they can use the values from parameters and other variables.

In the following example, the `Destination` variable determines the destination path using the `environment` and `concat` config functions:

```
{
    "Parameters": {
      "Source": { "Type": "string", "Description": "The source of files" }
    },
    "Variables": {
      "Destination": "[concat(environment('SystemDrive'),'\\newfiles')]"
    },
    "Tasks": {
      "CopyFiles" : {
        "Type": "Copy",
        "Params": {
          "Source": "[parameter('Source')]",
          "Destination": "[variable('Destination')]"
        }
      }
    }
}
```

3.1.5 Modules

The Sitecore Installation Framework provides many tasks and config functions. You can load additional tasks and config functions by including them in a configuration.

If the module is available on PSModulePath, you can load modules by name. Alternatively, you must load modules using an explicit path. Modules are loaded at the beginning of an installation.

If the additional features are packaged in a module on your computer, you can add them in the `Modules` section of a configuration to import them into the install session. For example:

- To load a module by name –`"MyCustomModule"`.
- To load a module by explicit path – `"C:\\extensions\\extensions.psm1"`.

```
{
    "Modules": [
      "MyCustomModule",
      "C:\\extensions\\extensions.psm1"
    ]
}
```

3.1.6 Uninstall Tasks

Configurations can optionally contain an `UninstallTasks` section. This section takes the same form as the `Tasks` section. It should contain a list of tasks that will undo all the actions completed in the `Tasks` section.

Uninstall tasks are invoked either by passing the `-Uninstall` switch to `Install-SitecoreConfiguration` or by calling the `Uninstall-SitecoreConfiguration` alias. It is not necessary to include the `-Uninstall` switch if you call `Uninstall-SitecoreConfiguration`.

Uninstall tasks can be treated in the same way as normal tasks and support all the same functionality as tasks.

```
{
    "Parameters" :{
        "SolrService": {
            "Type": "string",
            "DefaultValue": "Solr-7.2.1",
            "Description": "The name of the Solr service."
        },
    },
    "UninstallTasks": {
        "RemoveSolrService": {
            "Type": "RemoveService",
            "Params": {
                "Name": "[parameter('SolrService')]"
            }
        },
    }
}
```

In this example, the `RemoveSolrService` uninstall task executes the `RemoveService` task and stops and removes the service specified in the `SolrService` parameter.

If you have referenced other configurations in the `Includes` section, the `Uninstall Tasks` for each of the included configurations are run, in reverse order, after the tasks in the first configuration.

3.1.7 Register

Each entry in the `Register` section allows you to expose PowerShell functions as Sitecore Install Framework Tasks or ConfigFunctions within the config file for use in the `Variables` or `Tasks` sections. They are registered automatically with the `Register-SitecoreInstallExtension` command.

The section takes the format

```
{
    "Register": {
          "Tasks" : {
            "NewSMBShare" : "New-SMBShare",
            "Sleep": "Start-Sleep"
          },
          "ConfigFunction": {
            "GetRandom" : "Get-Random"
        }
    }
}
```

This configuration registers the `New-SMBShare` cmdlet as a task with the name `NewSMBShare` and registers `Start-Sleep` as a task with the name `Sleep`. The `Get-Random` cmdlet is registered as a Config Function with the name `GetRandom`.

As with extending the Sitecore Install Framework, the `Tasks` section of `Register` is for functions that don't return anything and the `ConfigFunctions` section is for functions that return a value.

3.1.8 Automatic Registration of Extensions.

Configuration files can optionally declare the `AutoRegisterExtensions` setting that allows dynamic registration of `Tasks` and `ConfigFunctions`. The default value is `false`.

Any PowerShell cmdlet can be referenced in a config file using a de-hyphenated version of its name.

```
{
    "Tasks": {
        "RandomSleep": {
            "Type": "StartSleep",
```

```
            "Params": {
                "Seconds": "[GetRandom(10)]"
            }
                }
    },
    "Settings" : {
        "AutoRegisterExtensions" : true
    }
}
```

In this configuration, the `RandomSleep` task automatically registers and executes `Start-Sleep` as a task and automatically registers and executes `Get-Random` as a `ConfigFunction`. It is the equivalent of running `Start-Sleep -Seconds (Get-Random 10)` on the command line.

If two or more cmdlets match, the cmdlets are listed and an error is thrown.

3.1.9 Includes

Configurations may refer to other configuration files to allow reuse of elements and reduce repetition in different configurations.

For example, the following config is a pseudo config which will configure a site. This is saved as `CreateSite.json`:

```
{
  "Parameters": {
    "Destination":{
      "Type": "string"
    },
    "SourcePackage":{
      "Type": "string"
    },
    "SiteName":{
      "Type": "string"
    }
  },
  "Tasks":{
    "CreateSite":{
      "Type": "CreateSite",
      "Params":{
        "SiteName": "[parameter('SiteName')]",
        "Path": "[parameter('Destination')]"
      }
    },
    "InstallPackage":{
      "Type": "CreateSite",
      "Params":{
        "Source": "[parameter('SourcePackage')]",
        "Path": "[parameter('Destination')]"
      }
    }
  }
}
```

Another config can be created which includes the previous example:

```
{
  "Includes":{
    "CreateSite":{
      "Source": ".\\CreateSite.json"
    }
  }
}
```

You can override elements of the included config by using the namespace it was imported as (in this case `'CreateSite'`):

```
{
  "Parameters": {
    "CreateSite:Destination":{
      "Type": "string",
      "DefaultValue": "C:\\inetpub"
    }
  },
```

```
  "Includes":{
    "CreateSite":{
      "Source": ".\\CreateSite.json"
    }
  }
}
```

This example overrides the default value for the destination parameter imported by the `CreateSite` config. A similar syntax is used to override `Variables` and `Tasks`.

We can also include the same config multiple times, as the name will be used to namespace each feature:

```
{
  "Parameters": {
    "CreateSite:Destination":{
      "Type": "string",
      "DefaultValue": "C:\\inetpub"
    },
    "CreateBackupSite:Destination":{
      "Type": "string",
      "DefaultValue": "C:\\inetpubbackup"
    }
  },
  "Includes":{
    "CreateSite":{
      "Source": ".\\CreateSite.json"
    },
    "CreateBackupSite":{
      "Source": ".\\CreateSite.json"
    }
  }
}
```

3.1.10 Settings

Settings let you configure the default requirements of the installation process. Some settings can also be overridden at runtime for the user by passing them as parameters to the `Install-SitecoreConfiguration` cmdlet.

For example:

```
Install-SitecoreConfiguration -WarningAction Stop -InformationAction SilentlyContinue.
```

The values in the settings are applied from lowest to highest in the following order:

- Default value – Contained in code.
- Configuration – Set in the configuration file.
- Command line – Passed at the command line.

Name	Default Value	Allowed Values	Command line?	Description

`WarningAction`	`Continue`	`Continue`
		`Ignore`
		`Inquire`
		`SilentlyContinue`
		`Stop`
		`Suspend`
			Yes	The action to take when a warning occurs.

`ErrorAction`	`Stop`	`Continue`
		`Ignore`
		`Inquire`
		`SilentlyContinue`
		`Stop`
		`Suspend`
			Yes	The action to take when an error occurs.

Name	Default Value	Allowed Values	Command line?	Description

`InformationAction`	`Continue`	`Continue`
		`Ignore`
		`Inquire`
		`SilentlyContinue`
		`Stop`
		`Suspend`
			Yes	The action to take when information is logged.

`AutoRegisterExtensios`	`False`	`True` `False`	No	Enables Dynamic registration of Tasks and Config Functions.

Settings within configuration files can be declared as:

```
{
    "Settings": {
        "AutoRegisterExtensions": true,
        "InformationAction": "SilentlyContinue"
    }
}
```

This configuration enables the `AutoRegisterExtensions` feature and sets the `InformationAction` preference to `SilentlyContinue`.

3.2 Create Tasks

Tasks are PowerShell functions that you can invoke from within a Sitecore Installation Framework configuration. When you invoke a configuration, each task performs an action. Because a task is implemented as a PowerShell function, it benefits from all the features that PowerShell offers.

3.2.1 The CmdletBinding Attribute

When you create a task, you must use the `CmdletBinding` attribute. This provides support for common PowerShell parameters, for example, those that control error handling.

It is best practice to use the `SupportsShouldProcess` parameter so that users can test the actions that the task takes, without applying them. The following example shows the use of the `CmdletBinding` attribute in the `Invoke-UnpackTask` cmdlet:

```
Function Invoke-UnpackTask {
    [CmdletBinding(SupportsShouldProcess=$true)]
    param(
      # Parameters
    )
    # function code
}
```

For more information about the `CmdletBinding` attribute, see the About Functions CmdletBindingAttribute article on the MSDN website.

3.2.2 Task Parameters

Task parameters are declared as normal PowerShell parameters. You can use validation and types to restrict the values that can be passed to the cmdlet. This includes marking parameters as mandatory, and support for multiple parameter sets.

When a task is called from a configuration, the *Params* property is mapped to the parameters declared in the PowerShell function. For example, the `Invoke-CopyTask` cmdlet declares the following parameters:

```
Function Invoke-CopyTask {
    [CmdletBinding(SupportsShouldProcess=$true)]
    param(
      [Parameter(Mandatory=$true)]
      [ValidateScript({ Test-Path $  })]
      [string]$Source,
      [Parameter(Mandatory=$true)]
      [ValidateScript({ Test-Path $  -IsValid })]
      [string]$Destination
    )
# function code
}
```

- The `Source` parameter is mandatory and checks that the given value is a string that points to a path that exists.
- The `Destination` parameter is also mandatory and checks that the given value is a string that is a valid file path.

The following is an example of how to declare the `Copy` task in a configuration where only the mandatory parameters are used:

```
{
    "Tasks": {
      "CopySomeFiles": {
        "Type": "Copy",
        "Params": {
          "Source": "c:\somefile\example.txt",
          "Destination": "c:\copied\"
        }
      }
    }
}
```

3.2.3 Return Values from Tasks

Tasks do not need return values. When invoked through a configuration, values returned from a task are not captured or processed directly. However, any value that you return from a task is shown in the install logs.

3.2.4 Write to the Logs

Previous versions os SIF used the `Write-TaskInfo` function to log information. This function has been deprecated and will be removed in a future version. Use the `Write-Information` function instead.

```
Write-Information -MessageData task "[Info] Updated"
```

Silent Output

Silent output can be achieved by re-directing all the streams to `$null`.

```
Install-SitecoreConfiguration -Path MyConfig.json *> $null
```

Creating a log file

In version 1.x, every time you used the built-in PowerShell transcript features to invoke `Install-SitecoreConfiguration`, a log file was automatically created.

This feature has been removed due to issues with different hosts and with the information that was logged to a log file.

To create a log file for an install we suggest the following syntax:

```
c:\> Install-SitecoreConfiguration <parameters> *>&1 | Tee-Object <logfile>
```

`*>&1` merges every stream - information, warning, and so on into the output stream.

`| Tee-Object <logfile>` outputs the stream to the console and a log file.

3.2.5 Include Tasks in a Configuration

Once a task has been written, it must be registered with the Installation Framework. Tasks can be included in a configuration by packaging them as a PowerShell module and adding them to the `Modules` section of a configuration, by directly registering them within a configuration, or enabling the `autoregisterextensions` function.

By using the `Register-SitecoreInstallExtension` cmdlet, the task is made available for use in configurations. For example, to register the `Copy-CustomItems` cmdlet as the `CustomCopy` task, use the following cmdlet:

```
Register-SitecoreInstallExtension -Command Copy-CustomItems -As CustomCopy -Type Task
```

Note
You can replace an existing registered task by using the `-Force` parameter. The following custom cmdlet replaces the default copy task: `Register-SitecoreInstallExtension -Command Copy-CustomItems -As Copy -Type Task -Force`.

3.3 Create Config Functions

Config functions are PowerShell functions that you can invoke from within a Sitecore Installation Framework configuration to access and calculate values that can be passed to a task.

Because a config function is implemented as a PowerShell function, it benefits from all the features that PowerShell offers. This includes parameter validation, strict mode, requires, and others.

Config functions must always return a value, and this value can be used by other config functions or by tasks within a configuration.

3.3.1 Config Function Parameters

Config function parameters are declared as normal PowerShell parameters. You can use validation and types to restrict the values that can be passed. This includes marking parameters as mandatory and support for multiple parameter sets.

When a config function is called from a configuration, the parameters are applied in the order that they are declared. If you want them to be applied in a different order, use the Position argument to specify this.

The `Invoke-JoinConfigFunction` function declares the following parameters:

```
Function Invoke-JoinConfigFunction {
    param(
        [Parameter(Mandatory=$true)]
        [psobject[]]$Values = @(),
        [Parameter(Mandatory=$false)]
        [string]$Delimiter = ","
    )

    # function code
}
```

In a configuration, this can be used as follows:

```
{
    "Parameters": {
      "Values": { "Type": "string[]", "DefaultValue": [ 1,2,3,4,5 ]
    },
    "Variables": {
      "Joined": "[join(parameter('Values'), '-')]
    }
}
```

When the `Joined` variable is evaluated, it results in a value of: `1-2-3-4-5`.

3.3.2 Include Config Functions in a Configuration

Once a config function has been written, it must be registered with the Installation Framework. Config functions can be included in a configuration by packaging them as a PowerShell module and adding them to the `Modules` section of a configuration.

By using the `Register-SitecoreInstallExtension` cmdlet, the config function is made available for use in configurations. For example the `Get-CustomJoin` cmdlet can be registered as the `CustomJoin` config function using the following cmdlet:

```
Register-SitecoreInstallExtension -Command Get-CustomJoin -As CustomJoin -
Type ConfigFunction
```

Note
You can also replace an existing registered config function by using the `-Force` parameter. The following replaces the default `join` config function with a custom cmdlet:

```
Register-SitecoreInstallExtension -Command Get-CustomJoin -As Join -Type ConfigFunction -Force
```

3.4 Invoking an Installation

To start a Sitecore installation the following syntax is used.

```
        Install-SitecoreConfiguration [-Path] <String> [[-Tasks] <String[]>] [[-From]
<String>] [[-To] <String>]
         [[-Skip] <String[]>] [[-WorkingDirectory] <String>] [-WhatIf] [-Confirm]
[<CommonParameters>]
```

This will begin a Sitecore installation using the given configuration loaded from a path.

The working directory is set as follows:

- If provided, the working directory is set to the given path.
- The current directory is used.

One or more tasks can also be passed to enable the execution of only a selection of tasks from the full configuration. If no tasks are passed (or an empty list is provided) all the tasks are executed. Execution of tasks can be further isolated by providing the `'From'` and `'To'` parameters. This executes an inclusive subset of tasks.

Parameters contained in the configuration file can also be overridden at the command line.

Example 1

```
PS C:\> Install-SitecoreConfiguration -Path .\MyConfig.json
```

Starts an installation based on a JSON configuration file.

Example 2

```
PS C:\> Install-SitecoreConfiguration -Path .\MyConfig.json -Tasks Alpha,Beta,Epsilon
```

Starts an installation based on a JSON configuration file and executes only the named tasks.

Example 3

```
PS C:\> Install-SitecoreConfiguration -Path .\MyConfig.json -Skip Alpha,Beta
```

Starts an installation based on a JSON configuration file and executes all the tasks except the named tasks.

Example 4

```
PS C:\> Install-SitecoreConfiguration -Path .\MyConfig.json -From Beta
```

Starts an installation based on a JSON configuration file and executes from the specified task.

Example 5

```
PS C:\> Install-SitecoreConfiguration -Path .\MyConfig.json -From Alpha -To Beta
```

Starts an installation based on a JSON configuration file and executes from the task named `Alpha` to the task named `Beta`.

Example 6

```
PS C:\> Install-SitecoreConfiguration -Path .\MyConfig.json -SiteName 'MySite'
```

Starts an installation based on a JSON configuration file, overriding the value for the `SiteName` parameter contained in that file.

Example 7

```
PS C:\> Install-SitecoreConfiguration -Path .\MyConfig.json -SkipValidation
```

Starts an installation based on a JSON configuration file and skips parameter validation.

Chapter 4

Further Guidance & Troubleshooting

The Sitecore Installation Framework contains embedded documentation about tasks and config functions that you can access directly from the PowerShell command line.

This chapter contains the following section:

- Further Usage and Help
- Troubleshooting

4.1 Further Usage and Help

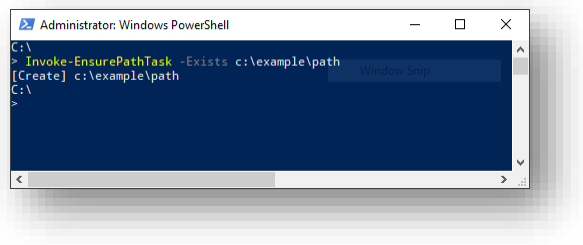
This section contains the following additional information that might be useful when you are using the Sitecore Installation Framework:

- Run Tasks and Config Functions Directly
- Execution Policies
- Get Help on the Installation Framework

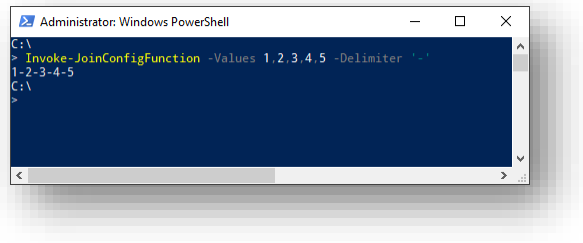
4.1.1 Run Tasks and Config Functions Directly

Both tasks and config functions are implemented as PowerShell cmdlets. When the Sitecore Installation Framework has been installed, you can directly invoke the cmdlets using standard PowerShell syntax.

Running tasks or config functions directly allows you to test the results at the command line. You can also integrate the commands into your own PowerShell scripts. For example, you can directly invoke the `EnsurePath` task by using its full PowerShell syntax:



Similarly, you can directly invoke the `Join` config function by using its full PowerShell syntax:



You can see the next chapter in this guide to get more help and guidance on using tasks and config functions directly in the PowerShell module.

4.1.2 Execution Policies

The execution policies in PowerShell let you restrict the conditions in which scripts and modules are loaded.

You can set policies for the computer, the user, or for the current session. You can also apply them by using a Group Policy. The Sitecore Installation Framework is digitally signed. This means that the module can be imported and executed in a PowerShell session running under any execution policy, except **Restricted**.

For more information about PowerShell Execution Policies, see the following link.

4.1.3 Get Help on the Installation Framework

The Sitecore Installation Framework contains information about each task and config function, and there is also documentation about the general use of the framework:

- `about_SitecoreInstallFramework`: contains general information about the framework.

- `about_SitecoreInstallFramework_Extending`: contains information about extension points that let you customize the framework for your specific installation needs.
- `about_SitecoreInstallFramework_Configurations`: contains examples of how to use tasks, scripts, and modules if you extend the framework.

There are three ways to view the help documentation:

- In the PowerShell window.
- With a markdown reader.
- As HTML pages.

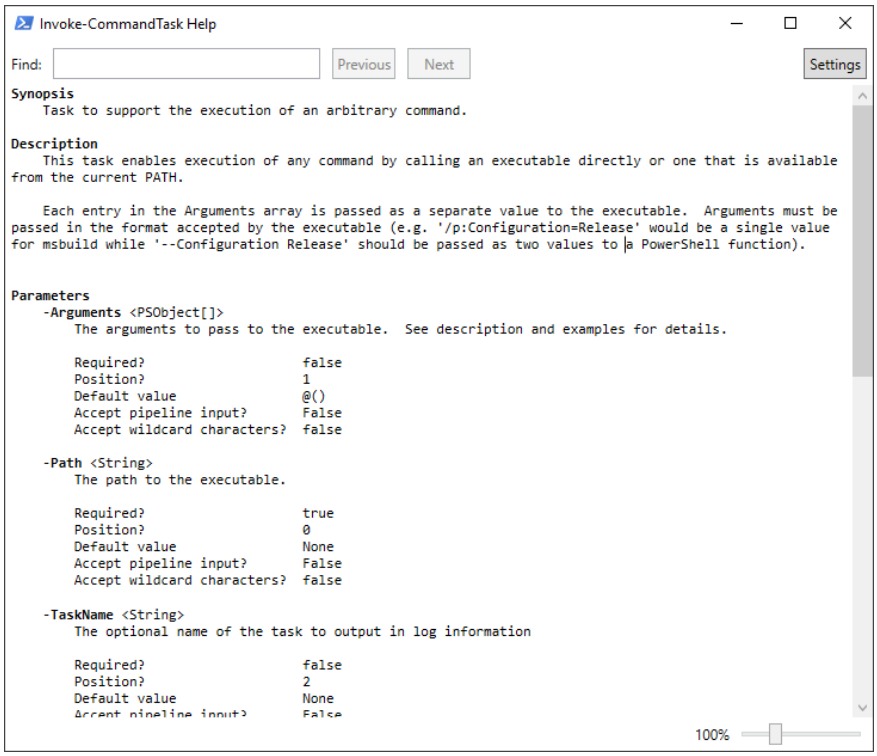
In addition to the help for individual tasks and functions, it is also possible to get a list of all available tasks or config functions using the `Get-Command` cmdlet.

View Help in the PowerShell Window

The PowerShell module has embedded help documentation that you can read.

To access the help for a specific PowerShell command or function:

- In PowerShell, enter the `Get-Help` cmdlet. When you do this, the help is displayed in the command line. If you want to open the help in a separate window, add the `ShowWindow` parameter to the `Get-Help` command. For example: `Get-Help Invoke-CommandTask -ShowWindow`



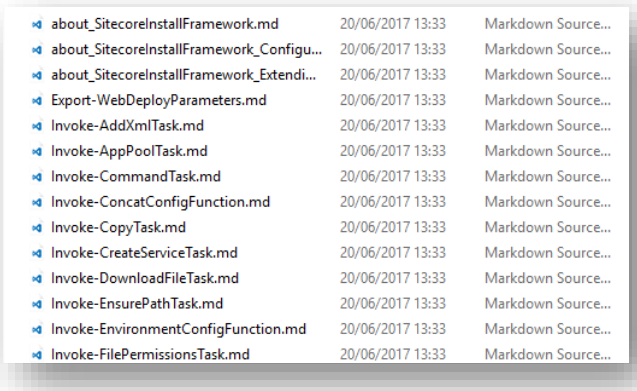
View Help with a Markdown Reader

When you unpack the Sitecore Installation Framework, it contains a folder of markdown documentation that you can read with any text editor or markdown reader.

To read the documentation:

- In Windows, navigate to the `SitecoreInstallFramework\docs` folder and use a text editor or markdown reader to open the relevant topic.

The following screenshot shows a subset of the topics that are available:



View Help as HTML Pages

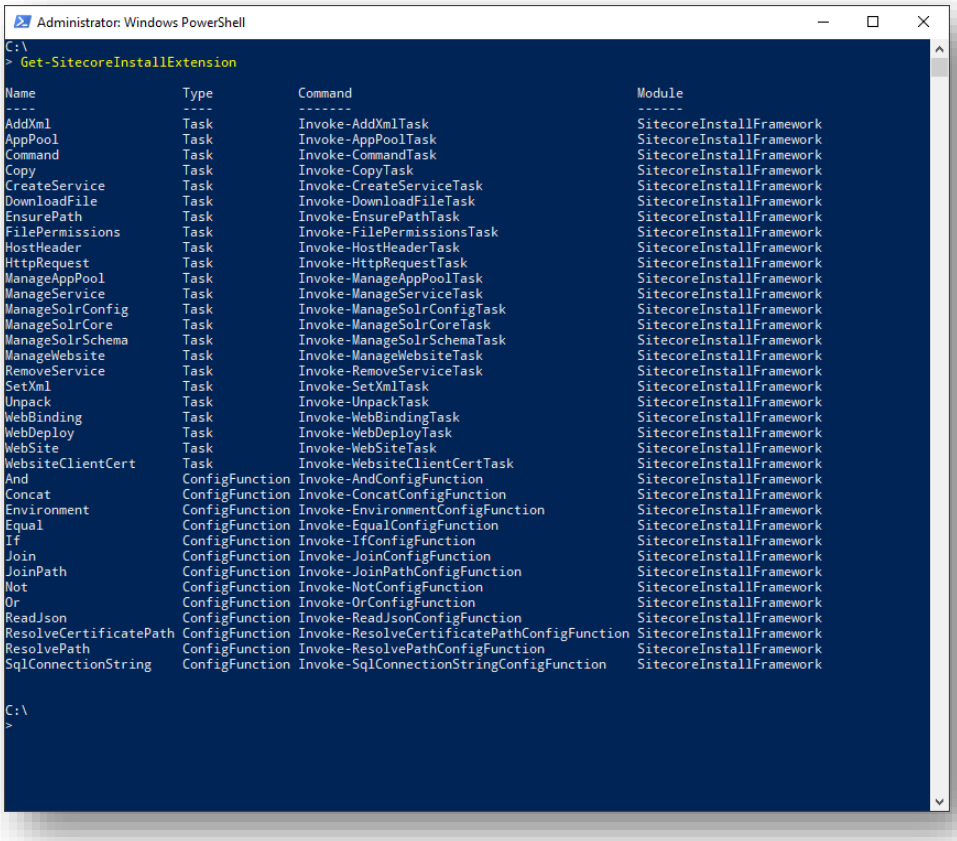
The Sitecore Installation Framework contains a folder of HTML pages that you can read with any browser.

To read the documentation:

1. In Windows Explorer, navigate to the `SitecoreInstallFramework\docs\html` folder.
2. To open it in a browser window, double-click the relevant topic.

Get a List of the Available Tasks and Config Functions

To see the tasks and config functions that are available, run the `Get-SitecoreInstallExtension` cmdlet.



You can filter the list by passing a value to the `Type` parameters. The following command only returns the tasks:

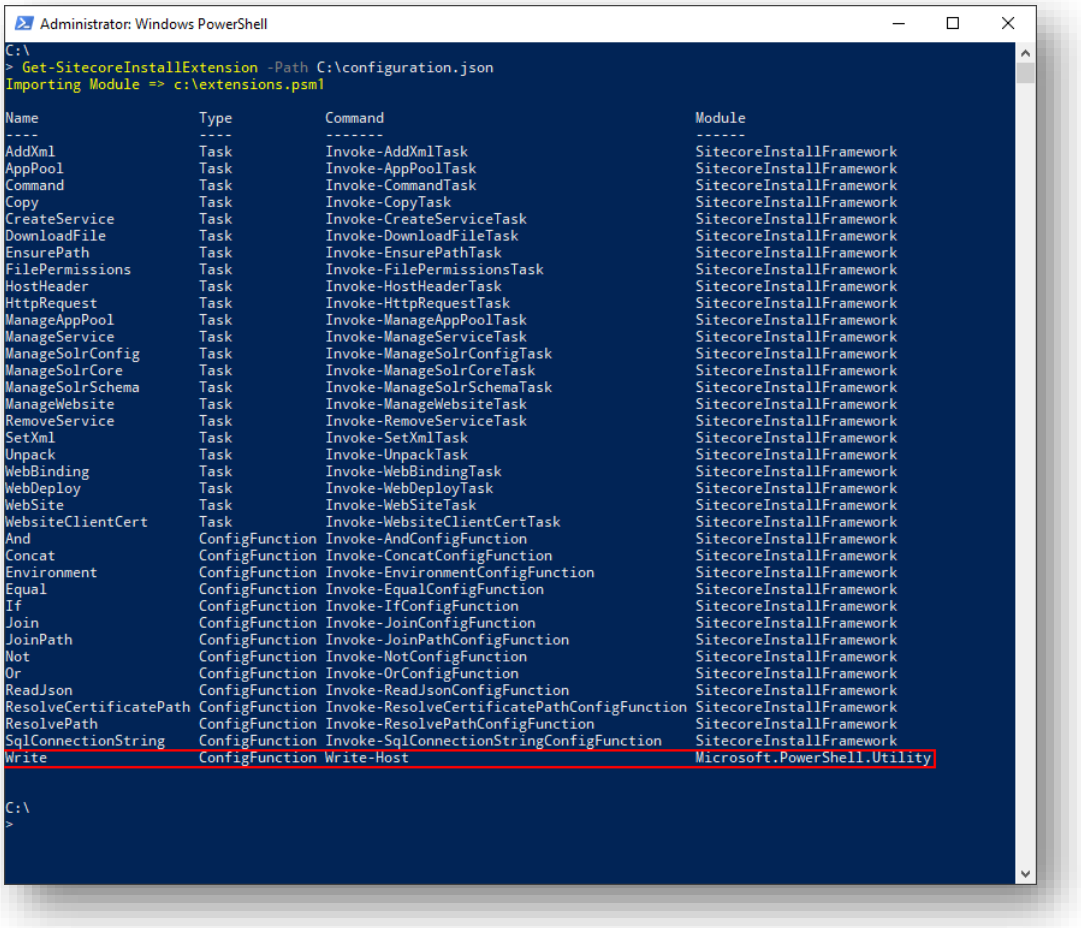
```
Get-SitecoreInstallExtension -Type Task.
```

To return the tasks and config functions that are available when a particular configuration is run, you must pass the path to the configuration file to the `Path` parameter.

For example,

```
Get-SitecoreInstallExtension -Path c:\configuration.json
```

displays the default tasks and config functions, as well as the extra registrations brought in by the configuration.



4.2 Troubleshooting

This section describes some of the issues you might encounter when using the Sitecore Installation Framework, and how to resolve them.

Internal Server Error

After a successful installation, the CM instance cannot be started and the following error is displayed:

```
HTTP Error 500.19 - Internal Server Error
The requested page cannot be accessed because the related configuration data for the page is
invalid.
Error Code 0x8007000d
```

To resolve this error, you must install the URL Rewrite module for IIS:

- Install URL Rewrite 2.1 using the Web Platform Installer.

Error When Invoking the Web Deploy Task

When you execute the Web Deploy task (`Invoke-WebDeployTask`), you can see the following error:

`ERROR_SCRIPTDOM_NEEDED_FOR_SQL_PROVIDER`

This error appears when the web deploy package installs databases using the SQL DACFx framework, and the provider has not been registered.

To resolve this error, ensure that you have the following components installed:

- SQL Server System CLR Types (2016 version)
- SQL Server Transact-SQL ScriptDom (2016 version)
- SQL Server Data-Tier Application (2016 version)

Note
If you are running the Sitecore installation from a 64-bit computer (x64), you must install both the 32-bit (x86) and 64-bit versions of the SQL Server components.

If you still receive the error after installing the SQL Server components, you must directly register the ScriptDom components.

To register the ScriptDom component:

1. Find the path to the ScriptDom library. In Windows Explorer, navigate to the folder `C:\Program Files (x86)\Microsoft SQL Server`.
2. The *Microsoft SQL Server* folder contains one or more subfolders. Click the subfolders (`\90`, `\100`, `\110`, `\120`, `\130`) and find the `\DAC\bin\Microsoft.SqlServer.TransactSql.ScriptDom.dll` file.
3. Copy or write down the path where the `.dll` file is located.
4. Launch PowerShell and navigate to the *NETFX 4.5.1 Tools* folder – `C:\Program Files (x86)\Microsoft SDKs\Windows\v8.1A\bin\NETFX 4.5.1 Tools`.
5. Invoke the `gacutil` application and enter the path that you copied in step 3. For example:

```
gacutil.exe /i C:\Program Files (x86)\Microsoft SDKs\Windows\v8.1A\bin\NETFX 4.5.1
Tools\120\DAC\bin\Microsoft.SqlServer.TransactSql.ScriptDom.dll
```

Missing Modules

Some features within the Sitecore Installation Framework require that other modules be loaded as well. You might see warnings that certain tasks cannot be loaded when importing the module.

You can continue to use other features in the module, however, the features that displayed warnings cannot be executed. For example, if the `WebAdministration` module is not available, you see the following warnings:

```
> Import-Module .\SitecoreInstallFramework
WARNING: The script 'Invoke-AppPoolTask.ps1' cannot be run because the following modules that are specified by the
"#requires" statements of the script are missing: WebAdministration.
WARNING: The script 'Invoke-WebBindingTask.ps1' cannot be run because the following modules that are specified by the
"#requires" statements of the script are missing: WebAdministration
WARNING: The script 'Invoke-WebsiteTask.ps1' cannot be run because the following modules that are specified by the
"#requires" statements of the script are missing: WebAdministration
```

When this happens, the module is loaded but the following tasks are not available:

- `Invoke-AppPoolTask`
- `Invoke-WebBindingTask`
- `Invoke-WebsiteTask`

Note
To install the `WebAdministration` module, you must first configure IIS on your computer. For more details and steps to configure IIS, see the *Configuring IIS* section in the Sitecore Experience Platform 9.1 Installation Guide that you can download from the Sitecore Downloads page – https://dev.sitecore.net.

Administrator Permissions

To run the Sitecore Installation Framework, you must run PowerShell as an Administrator.

Important
If you try to use the Sitecore Installation Framework in a non-admin window, you can see one of the following error messages and you will not be able to install Sitecore.

```
Import-Module : The required module 'SitecoreFundamentals' is not loaded. Load the module or
remove the module from 'RequiredModules' in the file
```

```
import-module : The script 'SitecoreFundamentals.psm1' cannot be run because it contains a
"#requires" statement for running as Administrator. The current Windows PowerShell session is
not running as Administrator. Start Windows PowerShell by  using the Run as Administrator
option, and then try running the script again.
```

My Sitecore installation failed while I was using Skype

- When using Skype while you are installing Sitecore 9.1, it is possible that your xConnect installation may fail. This is due to Skype and Sitecore xConnect using port 443, which interferes with the installation. If this happens, you must change your Skype configuration as outlined in the following article.
- Another approach to solving this installation issue is to update the default HTTP port, and if needed the HTTPS port.

Updating the default HTTP port

Updating the default HTTP port can be applied to the following Sitecore configurations:

- All Sitecore configurations
- All xConnect configurations

To update the default HTTP port:

1. In a text editor, open the relevant configuration file, for example: `sitecore-xp0.json`.
2. In the `CreateWebsite` task, add a `Port` property to the `Params` collection with the new value. For example:

```
"CreateWebsite": {
```

```
            // Creates or updates the IIS website instance.
            "Type": "Website",
            "Params": {
                "Name": "[parameter('SiteName')]",
                "ApplicationPool": "[parameter('SiteName')]",
                "PhysicalPath": "[variable('Site.PhysicalPath')]",
 "Port": 8080
            }
        }
```

Note
Ensure that you add a trailing comma to the line above the new property.

Updating the default HTTPS port

Certain configurations also configure secure bindings on the default HTTPS port (443). You can change this by updating the configuration with a different port. Updating the default HTTPS port can be applied to the following Sitecore configurations:

- All Sitecore configurations except `Sitecore-XM1-cd.json` and `Sitecore-XP0.json`
- All xConnect configurations

To change the default HTTPS port:

1. In a text editor, open the relevant configuration file, for example `sitecore-solr.json`.
2. In the `CreateBindingsWithThumbprint` task, add a `Port` property to the `Add` collection with the new value.
3. In the `CreateBindingsWithDevelopmentThumbprint` task, add a `Port` property to the `Params` collection with the new value. For example, the `CreateBindingsWithThumbprint` task can be updated to:

```
"CreateBindingsWithThumbprint": {
            // Configures the site bindings for the website.
            "Type": "WebBinding",
            "Params": {
                "SiteName" : "[parameter('SiteName')]",
                "Add": [
                    {
                        "HostHeader": "[parameter('SiteName')]",
                        "Protocol": "https",
                        "SSLFlags": 1,
                        "Thumbprint": "[variable('Security.SSL.CertificateThumbprint')]",
          "Port": 4443
                    }
                ]
            },
            "Skip": "[not(parameter('SSLCert'))]"
        }
```

Note
Ensure that you add a trailing comma to the line above the new property. For example, the `CreateBindingsWithDevelopmentThumbprint` can be updated to the following:

```
"CreateBindingsWithDevelopmentThumprint": {
            // Creates a new thumprint with a custom CA
            "Type": "AddWebFeatureSSL",
            "Params": {
                "HostName": "[parameter('SiteName')]",
                "OutputDirectory": "[variable('Site.DataFolder')]",
 "Port": 4443
            },
            "Skip": "[parameter('SSLCert')]"
        }
```