# Table of Contents

# RPS Packaging v3.1.0

*Last updated on June 28, 2021.*

## Welcome to the RPS v3.1.0 Packaging Documentation Landing Page

Rapid Provisioning System (**RPS**) is a flexible and powerful automation tool for managing software installation, updates, and configuration. This repository has been created specifically for RPS developers and other RPS administration patching roles.

## What is RPS Packaging Documentation?

RPS Packaging documentation provides details about the RPS packaging system, including how to properly operate and maintain RPS; and how to load, deploy, and check the status of patches and ISOs in RPS.

⚠ **IMPORTANT**

Documentation bundled with RPS v3.1.0 is accurate as of **1/15/2021**.

Updated documentation can be found at: https://reactr.azurewebsites.us

To access documentation on how to create, edit, delete, and download patches and ISOs in REACTR, please visit RPS Patching in REACTR, located at the website above.

# Access Control for Patch Management and Sync

## Introduction

This document describes the account(s) and permissions required for Sync and Patching in Rapid Provision System (RPS).

### Document Overview

Access Control for Patch Management and sync is a document that specifies the account(s) and permissions required by RPS. This document is considered a living document and subject to change.

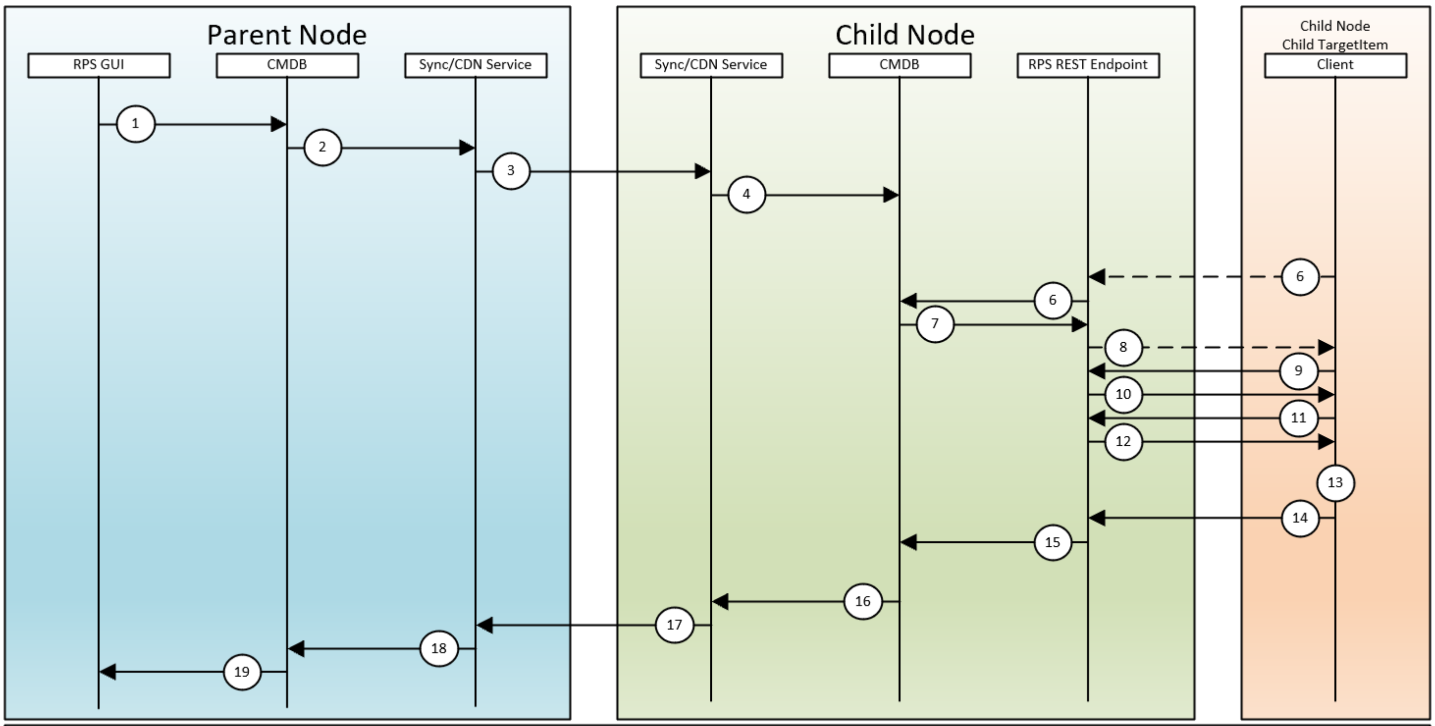
## Access Control

### Sync

RPS leverages a user account as a service account (with role SyncServiceAccount). SyncServiceAccount is the account used to collect and investigate which files it needs to download for BITS.
The SyncServiceAccount account is also used to configure DFSR. This account must be a member of the DFSRAdministrators group.

### Patch Management

Being that Patch Management in RPS leverages certificates to authenticate to the RPS feed and DSC, which runs on the local system account, it does not require any additional service accounts.



1. Package and Folder to Target assignments are made by approving a Package Stream and are stored as Resource assignments in the CMDB
2. Updated resource assignments are detected by the Sync Service
3. Updated resource assignments are transmitted to child nodes
4. Child node Sync Service updates local CMDB with new resource assignments
5. Client DSC configuration sends request to REST endpoint with configured Target Item ID
6. REST Endpoint queries CMDB for all Resource Assignments of type Package for the specified Target ID
7. Resource Assignment objects are returned to the REST endpoint
8. List of Package name/version, Ensure states, and deploy attempt counters are returned to client
9. Client then sends request for specific packages metadata
10. REST endpoint provides metadata for client to perform state check
11. Client then sends request for missing packages that are needed to reach desired state and request for Target Maintenance Windows
12. REST endpoint returns Packages and Maintenance Windows for install process
13. If Target is within Maintenance Window then the Package is deployed to the Target
14. Client sends state report to REST Endpoint
15. REST Endpoint updates resource assignment's 'InDesiredState', 'DeployedOn', 'PackageProcessedCounter', 'IsInMaintenanceWindow', and 'DeployedStatus' Properties based on received report data.
16. CMDB updates are detected by the Sync Service
17. Updates are transmitted to the parent node Sync Service
18. Parent node Sync Service updates local CMDB
19. GUI telemetry data is updated for each Package and Target.

# Sync Service Settings

These settings live within the Sync Service's configuration (app.config) file. Any changes to these settings would require a restart of the Sync Service in order to take effect. It is assumed that the reader has a knowledge of how time requirements as well as the specific functionality desired for a specific instance of RPS.

## General

The configuration settings inside the Sync Service application configuration file are located in the <appSettings> section of the document. This section will define each setting purpose and the valid values for the setting.

#### ContextCommandTimeout

*Type*: Integer

*Default*: 3600

*Range*: Any integer greater than or equal to 1. Recommended are value is 3600

Timeout, in seconds, for database calls from the Sync Service. Changing this value would alter the maximum amount of time a database call is allowed to execute. This prevents locking the database for abnormal amounts of time.

#### HttpClientTimeout

*Type*: Integer

*Default*: 3600

*Range*: Any integer greater than or equal to 1. Recommended are value is 3600

Timeout, in seconds, for HTTP(s) calls from the Sync Service. Changing this value would alter the amount of time a connection is allowed to be alive for a single HTTP(s) call. This would affect both the Sync operations as well as the requests to the Fileserver when synchronizing CDN packages

#### CreateClient

*Type*: Boolean

*Default*: True

Indicates whether or not to create a client to used to synchronize data changes with to and from the current node’s parent. The type of data synchronized when this setting is set to true includes target data, resources and properties, patch stream telemetry, etc. If this setting is set to false, all synchronization both to and from the parent will be disabled.

## Logging

*The following settings are for setting specific levels of Sync Service application logging.*

#### SyncLogs

*Type*: Boolean

*Default*: True

*Remarks*: If the CreateClient setting is off, these logs will not synchronize because data synchronization is turned off.

Indicates whether or not to Synchronize logs with the parent Sync Service. Setting the value to true will cause the logs on the client to synchronize with its parent. Setting the value to false will disable this synchronization.

#### LogLevel

*Type*: Integer

*Default*: 4

Indicates the level of logging the sync service will use when synchronizing the logs to the parent node. When a log is made in RPS, it is assigned a log level. When the synchronization of logs occurs, it will send all logs with a level equal to and higher than the value set in this setting to the parent.

1. Verbose – Setting this as the value will synchronize everything logged by the Sync service
2. Debug – Contains information useful for debugging.
3. Information – Contains general information.
4. Warning – Contains warnings for the user to take note of. General indicates that a potential error may occur due to some action taken, but its not guaranteed that it will result in an error.
5. Error – Contains information about an error that the Sync Service encountered.
6. Fatal – Contains information about a catastrophic error that usually results in the partial or total loss of the Sync Service.

## API Server

This section covers settings that are specific to the rest service endpoints for the Sync Service.

#### CreateServer

*Type*: Boolean

*Default*: True

Indicates whether or not to start a server instance to host the Sync Api endpoints. This server is needs to be enabled on both nodes in order to allow communications between them for both Syncing of data and CDN items. Setting this value to true will start the REST API server internally by the Sync Service. By this value to false, calls into the SyncService via remote sources would fail.

## CDN Settings

This section contains settings for the content delivery network.

#### CreateCDN

*Type*: Boolean

*Default*: True

Indicates whether or not to enable the CDN indexer functionality of the Sync Service. By this option this option to true, you will turn on the CDN and any subsequent synchronization of the CDN with a parent CDN. Setting this value to false will not start the CDN and its synchronizations, thus stopping any new BITS transfers from occurring. This setting has no ties to enabling the file system or api server.

#### IndexerInterval

*Type*: Integer

*Default*: 3

*Range*: Any integer greater than or equal to 1. Recommended are value is 3

The interval, in minutes, that the CDN will synchronize. A larger value will increase the time between synchronizations. CDN synchronization will look at what files are currently on the system vs what files should be there. If the system is set to BITS, it will then request any missing files from its parent and begin the download.

## CDN File Server

This section covers settings that are specific to the Content Delivery Network (CDN) File Server. The CDN File Server is the mechanism that transmits files to parent and child nodes.

CreateStaticFiles

*Type*: Boolean

*Default*: True

Indicates whether or not to start the FileServer in the Sync Service. This is required if the current node is expected to host CDN packages and distribute them downstream. In order to enable this service both the CreateServer setting must be true AND this setting must be true. That is because the file server endpoint is hosted in the API Server. Assuming the CreateServer setting is enabled, setting this value to true will allow for the downstream nodes to request packages from the current node. By setting this value to false, any requests for CDN packages by a child node will fail, stating the endpoint is not there (404 error).

FileServerOptions.RequestPath

*Type*: String

*Default*: /files

*Remarks*: The value MUST start with a "/" and must be a valid url

The part of the url that allows access to the CDN File Server where CDN files will be available from Sync Service. For example if the base was *www.contoso.com*, and the request path was */files*, the endpoint to access the file server *www.contoso.com/files*.

FileServerOptions.EnableDirectoryBrowsing

*Type*: Boolean

*Default*: True

Indicates whether or not to allow directory browsing of the file server. Setting this option to true will allow an individual to hit the root file server endpoint and get the directory view for all static files (CDN files) on the node. Setting the value to false disables this ability.

FileServerOptions.EnableDefaultFiles

*Type*: Boolean

*Default*: False

Indicates whether or not to enable default files on the file server. When this option is set to true, the server will attempt to try to server out an index.html file when visiting the file server root. Setting this value to false will stop the attempt to serve out the index.html file.

# Sync Testing

## General

The goal of this testing is to verify correct data replication between nodes in various scenarios. See the below section for an overview of the sync scenarios and concepts which are tested.

## Testing Scenarios

This section contains the scenarios for the Sync Service that are continually tested through the DevOps pipleline.

### Synchronizing Upstream and Downstream Changes

Validates that downstream and upstream changes are created at the appropriate times (not creating upstream changesets when **only** downstream changes occurred, and vice versa). However, sync is a bi-directional process, and both Nodes will evaluate what changes need to be merged when a sync occurs. These tests verify the proper creation and use of changesets between two Nodes.

### Sync Versioning

Each Node keeps track of its *SyncReceivedVersion* and *SyncSentVersion*. These two Node properties are used to determine when, and with who, a Node last synced. Additionally, Nodes also track a list of recent committed versions. This list can be used to obtain the previous sync version of a subscriber from a distributor (the distributor being the Node where the changes were made, and the subscriber the Node in need of the changes). The unit tests verify that this list of versions is maintained properly throughout synchronization.

### Sync Scope

These tests validate the correct adherence to *SyncScopes* with respect to the properties of entities such as TargetItems, ResourceItems, etc. The five different *SyncScopes* are as follows:

- **Public** -> Will Sync
- **Private** -> Will NOT Sync
- **Internal** -> Syncs to internal Nodes only
- **InternalDownStream** -> Syncs to internal children only
- **InternalUpstream** -> Syncs to internal parents only

### Synchronizing Target Data

Validates that TargetItems and TargetGroups only sync in one of two circumstances - always sync upstream, and only sync downstream when the *NodeId* matches that of the target data.

### Synchronizing Task Assignments

Validates that Tasking data is correctly synced both upstream and downstream. TaskMaps will sync both ways as long as the corresponding TargetItem syncs. This is the same for TaskFilters, TaskAssignments, and dependencies. Any changes made to Tasking data, such as updates and deletes, are also synced bi-directionally between Nodes. TaskItems with protected properties are also synced, and the protected properties should return an *IsProtected* flag.

### Synchronizing Create, Read, Update, and Delete

Validates the synchronization of CRUD operations across multiple tiers (Master -> Region -> Site 1 & 2).

### Synchronizing Resource Data

Validates the correct synchronization of resource data both upstream and downstream. Resource data, such as ResourceItems, always sync upstream. However, ResourceItems will only sync downstream if they are marked as Global, or assigned to a target on a child node. The following cases are tested:

- Synchronization of ResourceItems, ResourceGroups, and ResourceAssignments both upstream and downstream when applicable.
- The correct handling of empty ResourceGroups (which should not sync).
- Synchronization of new ResourceItem properties, as well as deleted properties.

### Conflict Behavior

The following Sync conflicts and corresponding resolutions are tested:

- Duplicate TargetItem name and type should rename source TargetItem
- Duplicate TargetGroup name and type should rename source TargetGroup
- Duplicate ResourceItem name and type should rename source ResourceItem
- Duplicate ResourceGroup name and type should rename source ResourceGroup

- Duplicate TargetItem property name should rename source property item
- Duplicate TargetGroup property name should rename source property item
- Duplicate ResourceItem property name should rename source property item
- Duplicate ResourceGroup property name should rename source property item
- Duplicate TaskItem should rename source item

*The entity is renamed by appending a timestamp to the current name*.

### Synchronizing Logging

Validates the correct synchronization of logs upstream depending on log level (inclusion and exlcusion). Logs do not sync downstream.

### Sync Dependency Order

Verifies proper synchronization of dependent CRUD operations (i.e inserting a TargetItem on a Node, and subsequently deleting it). Additionaly, these tests also validate that certain operations such as *Delete* are tracked based on the Sync version.

# Packaging Script Framework

The packaging script framework lets you consume a PowerShell script as a package.

## Use Cases

- Installing a package on an appliance, i.e. Firewall, where PowerShell cannot directly run on it.
- Creating a PowerShell script to install a package that has custom Post or Pre steps during installation.
    - Configure custom registry values for configuration.
- Installing a package that doesn't have an installer. i.e. just copying files to a location.

## Requirements of the Script Framework

The script framework consist of a PowerShell module and a manifest file zipped together. The zip structure may also contain other supporting files and scripts as required by the custom code.

## The PowerShell module

### Required Functions

There are three required functions that must exist in the PowerShell module.

| FUNCTION NAME | DESCRIPTION | RETURN TYPE |
|---|---|---|
| **Test-PackageResource** | This function test if the package is in desired state. | Must return a boolean. True for in desired state and false for not in desired state. |
| **Set-PackageResource** | This function does the install or uninstall. | None |
| **Get-ParameterMapping** | This function helps with complex mappings using Rps-Mapped Parameters. | Function will return a Hashtable for each custom parameter it needs to map. |

```
function Get-ParameterMapping
{
    return @{
            DscEncryptionCertificate = @{
                        EntityClass = 'ResourceItem'
                        EntityType = 'Certificate'
                        Role = 'DscEncryption'
                        IsAssigned = $true
            }
    }
}
```

Your PowerShell module can include other supporting functions and scripts as required for your package.

### Parameters

Test-PackageResource and Set-PackageResource only has one required parameter, Ensure. Ensure states if the package should be 'Present' or 'Absent'. The two methods can have any other parameters that are required for the custom script to run using Rps Mapped Parameters.

**NOTE**

For more information on RPS-Mapped Parameters see How to configure Rps-Mapped Parameters.

Example Script Framework

```
$null = Import-Module -Name 'VMware.PowerCLI'
$null = Set-PowerCLIConfiguration -InvalidCertificateAction Ignore -Confirm:$false -ErrorAction
SilentlyContinue -DefaultVIServerMode Multiple -ParticipateInCEIP $false -Scope Session

function Test-PackageResource
{
    [CmdletBinding()]
    Param
    (
        [Parameter(Mandatory = $true)]
        [ValidateScript({[ipaddress]::Parse($_)})]
        [string]
        $IPAddress,

        [Parameter(Mandatory = $true)]
        [string]
        $ComputerName,

        [Parameter(Mandatory = $true)]
        [PSCredential]
        $LocalAdmin,

        [Parameter(Mandatory = $true)]
        [string]
        $Ensure
    )

    Write-Verbose "Connecting to ESX at $IPAddress"

    $server = Connect-ViServer -Server $IPAddress -Credential $LocalAdmin -WarningAction SilentlyContinue -
ErrorAction Stop

    $virtualSwitches = Get-VirtualSwitch -Name 'PatchedVirtualSwitch' -ErrorAction SilentlyContinue
    if($virtualSwitches)
    {
        if ($ensure -eq 'present')
        {
            return $true
        }
        else
        {
            return $false
        }
    }

    if ($ensure -eq 'Absent')
    {
        return $true
    }
    return $false

    Write-Verbose "Connected to ESX at $IPAddress"
}

function Set-PackageResource
{
   [CmdletBinding()]
    Param
    (
        [Parameter(Mandatory = $true)]
        [ValidateScript({[ipaddress]::Parse($_)})]
        [string]
        $IPAddress,

        [Parameter(Mandatory = $true)]
        [string]
        $ComputerName,

        [Parameter(Mandatory = $true)]
        [PSCredential]
        $LocalAdmin,

        [Parameter(Mandatory = $true)]
        [string]
        $Ensure

    )
    $server = Connect-ViServer -Server $IPAddress -Credential $LocalAdmin -WarningAction SilentlyContinue -
ErrorAction Stop
    if ($ensure -eq 'Present')
    {
        New-VirtualSwitch -Server $server -Name 'PatchedVirtualSwitch' -ErrorAction SilentlyContinue
    }
    else
    {
        $vs = Get-VirtualSwitch -Server $server -Name 'PatchedVirtualSwitch'
        $vs | Remove-VirtualSwitch -Server $server -ErrorAction SilentlyContinue -Confirm:$false
    }
    Write-Verbose "Connecting to ESX at $IPAddress"
}

# Sample parameter mapping.  We are not using any complex parameters in this script
function Get-ParameterMapping
{
    @{
        DscEncryptionCertificate = @{
                    EntityClass = 'ResourceItem'
                    EntityType = 'Certificate'
                    Role = 'DscEncryption'
                    IsAssigned = $true
        }
    }
}
```

# RPS Packaging Manifest

*Last updated on April 20, 2021.*

This document describes the Packaging Manifest of the Rapid Provisioning System (RPS).

## What is the Package Manifest File

When creating the ZIP file that houses all the data needed to install a package, there also needs to be a package manifest file created with the package. The manifest file lives at the root level of the Zip Archive and has a name of "Package.RPS". This file contains the information such as the Operating System (OS) Type, OS Architecture, Package name, etc. This information is used by RPS to apply the patches and to determine what targets are to receive the packages.

## Structure of the Package Manifest File

The internal structure of the Package Manifest file is XML with a predetermined set of elements. The values contained in these elements will be the details applied to the package to be used to transfer, apply, and manage the packages on an RPS enabled system.

```
<?xml version="1.0" encoding="utf-8"?>
<PackageManifest version="1.0">
  <PackageName></PackageName>
  <PackageVersion></PackageVersion>
  <PackageClassification></PackageClassification>
  <OsVersion></OsVersion>
  <Architecture></Architecture>
  <OsType></OsType>
  <ProductName></ProductName>
  <ProductType></ProductType>
  <ProductVersion></ProductVersion>
  <ProductId>ProductId
  <InstallerFileName> </InstallerFileName>
  <SupressReboot></SupressReboot>
  <Products />
  <UninstallArguments> </UninstallArguments>
  <InstallArguments />
  <Description> </Description>
  <MsCatalogProductName> </MsCatalogProductName>
  <MsCatalogTitle />
  <MsCatalogId />
  <MsCatalogUpdateId />
  <MsCatalogSupercededByKbIds />
  <MsCatalogLinkUrls />
</PackageManifest>
```

## Package Manifest XML Attributes

*Version Attribute*

**Type:** String

**IsRequired:** Yes

This is the version of the package manager that will process this manifest file.

Currently the following versions exist:

- 1.0

```
<PackageManifest version="1.0">
```

## Package Manifest XML Elements

*PackageName Element*

**Type:** String

**IsRequired:** Yes

This is the name of the package to be used by RPS. This name is used in combination with the PackageVersion element to create a unique name.

*PackageVersion Element*

**Type:** String

**IsRequired:** Yes

This is the version of the package to be used by RPS. This name is used in combination with the PackageName element to create a unique name. The value stored in here typically follows Semantic Versioning.

*PackageClassification Element*

**Type:** String

**IsRequired:** Yes

The classification of the package. The acceptable values should be from a preset list of:

- Security
- Critical
- Definition
- New
- Script
- General

*OsVersion Element*

**Type:** String

**IsRequired:** Yes

The Version of the operating system to whom the patch is applicable. A wildcard (*) can be passed into target "any" OS version. Partial wildcard matching is also allowed, which means it will match the values before and after the wildcard. For example:

- 10.* would match 10.0.0.2, 10.2, etc because they all start with "10."
- *.1 would match 10.0.1, 10.1, etc because it they all end with ".1"

**NOTE**

The package will only be assigned to Targets who have an OsVersion property that passes the match test on the value provided in this field in addition to the Architecture and OsType values.

*Architecture Element*

**Type:** String

**IsRequired:** Yes

The architecture of the operating system to whom the patch is applicable. A wildcard (*) can be passed into target "any" OS version. Acceptable values are:

- x86
- x64
- *

> **NOTE**
>
> The package will only be assigned to Targets who have an Architecture property that passes the match test on the value provided in this field in addition to the OsVersion and OsType values.

*OsType Element*

**Type:** String

**IsRequired:** Yes

The type of the operating system. Currently these values should be Windows or Linux.

> **NOTE**
>
> The package will only be assigned to Targets who have an OsType property that passes the match test on the value provided in this field in addition to the OsVersion and Architecture values.

*ProductType Element*

**Type:** String

**IsRequired:** Yes

The ProductType of the package. This value will be to determine the type of package being installed.

Acceptable values are:

- WindowsHotfix
- WindowsMsi
- WindowsExe
- WindowsCabinet
- ScriptFramework
- LinuxRpm

*ProductName Element*

**Type:** String

**IsRequired:** Conditionally, if ProductType is WindowsMsi or WindowsExe

The name of the product. This value will be used when registering the product with the OS, if needed.

*ProductVersion Element*

**Type:** String

**IsRequired:** Conditionally, if ProductType is WindowsMsi or WindowsExe

The Version of the product. This value will be used when registering the package with the OS, if needed.

*ProductId Element*

**Type:** String

**IsRequired:** Conditionally, if ProductType is WindowsMsi, WindowsExe, WindowsHotfix, or WindowsCabinet

The ID of the product. This value will be used when registering the product with the OS, if needed.

*InstallerFileName Element*

**Type:** String

**IsRequired:** Yes

The name of the file inside of the package zip archive that should be executed to begin the install/uninstall process. This file should live on the root of the package.

*SupressReboot Element*

**Type:** Boolean (True/False)

**IsRequired:** Yes

Whether or not that the machine should reboot after installing/uninstalling the package.

*Products Element*

**Type:** Collection of Strings

**IsRequired:** No

A collection of products a package affects/supports. For example, a Windows update might patch SQL server, Windows 10, and Visual Studio.

```
<Products>
    <String>SQL Server</String>
    <String>Windows 10</String>
    <String>Visual Studio</String>
</Products>
```

*Conditions Element*

**Type:** Collection of PackageAssignmentCondition

**IsRequired:** No

A collection of custom conditions for filtering package assignments to a target.

```
<Conditions>
    <PackageAssignmentCondition>
      <Property>IsApp</Property>
      <Operator>Eq</Operator>
      <Value>True</Value>
    </PackageAssignmentCondition>
  </Conditions>
```

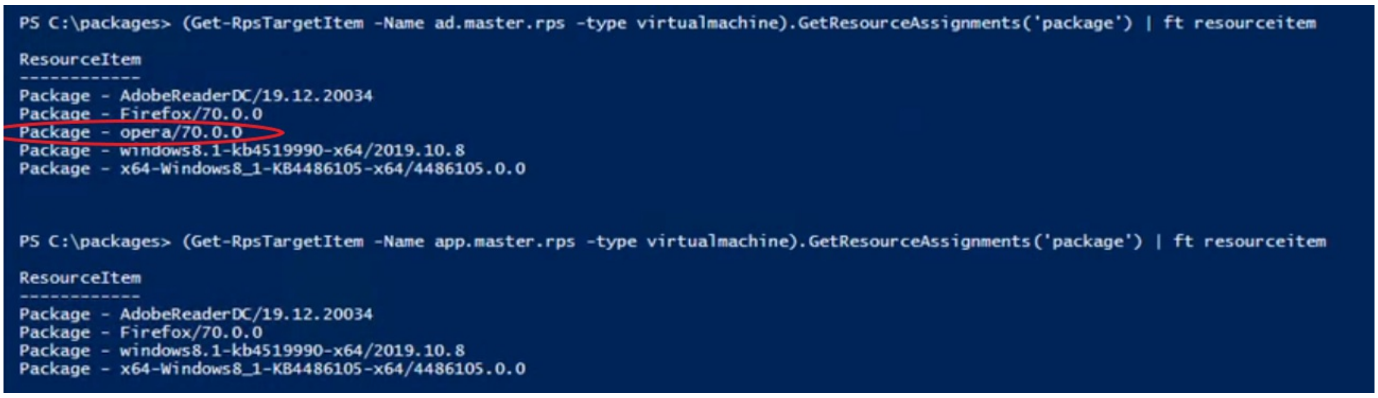
Known Issue

Package Manifest **Conditions** element **Value** field does not support multiple values separated by the pipe delimiter **|**

- **Error Details:** The following PackageManifest code snippet is an example using pipe delimiter **|** in **Conditions**, which will fail:

```
<InstallerFileName>opera.msi</InstallerFileName>
<Conditions>
  <PackageAssignmentCondition>
    <Property>Name</Property>
    <Operator>Eq</Operator>
    <Value>AD.master.rps|APP.master.rps</Value>
  </PackageAssignmentCondition>
</Conditions>
```

**The resulting behavior:** Only the first Value listed will be assigned to; all other Values after the pipe delimiter **|** are ignored.



```
PS C:\packages> (Get-RpsTargetItem -Name ad.master.rps -type virtualmachine).GetResourceAssignments('package') | ft resourceitem

ResourceItem
------------
Package - AdobeReaderDC/19.12.20034
Package - Firefox/70.0.0
Package - opera/70.0.0
Package - windows8.1-kb4519990-x64/2019.10.8
Package - x64-Windows8_1-KB4486105-x64/4486105.0.0


PS C:\packages> (Get-RpsTargetItem -Name app.master.rps -type virtualmachine).GetResourceAssignments('package') | ft resourceitem

ResourceItem
------------
Package - AdobeReaderDC/19.12.20034
Package - Firefox/70.0.0
Package - windows8.1-kb4519990-x64/2019.10.8
Package - x64-Windows8_1-KB4486105-x64/4486105.0.0
```

In this particular example, AD.master.rps is assigned the opera Package, because it was listed before the pipe delimiter **|** . APP.master.rps is not assigned the opera Package, because it was listed after the pipe delimiter **|** .

- **Current Workaround for pipe delimiter | :** Utilize the Match Operator `<Operator>Match</Operator>`, with each value in the Value field wrapped in parentheses **()** and with a trailing question mark **?** . Example:

```
<Conditions>
    <PackageAssignmentCondition>
        <Property>ComputerName</Property>
        <Operator>Match</Operator>
        <Value>(NFA)?(WNM)?(WNMA)?</Value>
    </PackageAssignmentCondition>
</Conditions>
```

In this particular example, a Target with a Property of ComputerName will be assigned if its Value contains NFA, WNM, **and/or** WNMA. This implementation only requires a **partial** Value match.

For an **exact** Value match, the full string in the Value field must be enclosed with a caret **^** and a dollar sign **$** . Example:

```
<Conditions>
    <PackageAssignmentCondition>
        <Property>ComputerName</Property>
        <Operator>Match</Operator>
        <Value>^(NFA)?(WNM)?(WNMA)?$</Value>
    </PackageAssignmentCondition>
</Conditions>
```

In this particular example, a Target with a Property of ComputerName will be assigned if its Value contains NFA, WNM, **and** WNMA.

*UninstallArguments Element*

**Type:** String

**IsRequired:** No

Arguments passed into when executing the InstallerFileName in order to uninstall the package.

*InstallArguments Element*

**Type:** String

**IsRequired:** No

Arguments passed into when executing the InstallerFileName in order to install the package.

*Description Element*

**Type:** String

**IsRequired:** No

A description of the package and what it's installing.

*MsCatalogProductName Element*

**Type:** String

**IsRequired:** No

The catalog product name of the package. Used/Populated by updates that come from a Microsoft Catalog.

*MsCatalogTitle Element*

**Type:** String

**IsRequired:** No

The catalog title of the package. Used/Populated by updates that come from a Microsoft Catalog.

*MsCatalogId Element*

**Type:** String

**IsRequired:** No

The catalog ID of the package. Used/Populated by updates that come from a Microsoft Catalog.

*MsCatalogUpdateId Element*

**Type:** String

**IsRequired:** No

The catalog update ID of the package. Used/Populated by updates that come from a Microsoft Catalog.

*MsCatalogSupercededByKbIds Element*

**Type:** Collection of Strings

**IsRequired:** No

The catalog IDs of KBs that supersede the package. Used/Populated by updates that come from a Microsoft Catalog.

```
<MsCatalogSupercededByKbIds>
    <String>4532693</String>
    <String>4532695</String>
    <String>4528760</String>
</MsCatalogSupercededByKbIds>
```

*MsCatalogLinkUrls Element*

**Type:** Collection of Strings

**IsRequired:** No

The catalog link URLs for the package. Used/Populated by updates that come from a Microsoft Catalog.

```
<MsCatalogLinkUrls>
    <String>https://www.catalog.update.microsoft.com/ScopedViewInline.aspx?updateid=db57861b-e22b-4107-8c78-
1ae8d63310d2</String>
    <String>https://www.catalog.update.microsoft.com/ScopedViewInline.aspx?updateid=567d7a0d-7f11-4c75-ba80-
e7dd1b88fbe3</String>
    <String>https://www.catalog.update.microsoft.com/ScopedViewInline.aspx?updateid=5f46c0b9-c57c-484e-b6e5-
80dded34bfa3</String>
</MsCatalogLinkUrls>
```

## Get-RPSPackageManifest

The cmdlet Get-RPSPackageManifest will return the XML Schema for the package manifest file. The cmdlet takes one optional parameter **Version**. The only valid value that is currently supported is '1.0'. Without the version parameter, this cmdlet will return the most current version of the manifest.

```
$currentSchema = Get-RPSPackageManifest
```

## Test-RPSPackageManifest

The cmdlet Test-RPSPackageManifest will take the path to a package manifest file and validate the file XML format. The cmdlet takes two parameters: PackageFilePath and, optionally, SchemaVersion. The PackageFilePath parameter is the path to the package manifest file to be validated. The optional SchemaVersion is used to specify a package manifest version to validate against. Currently only '1.0' is a valid value. If the SchemaVersion is not provided, the latest version of the manifest schema will be used to perform the validation.

If the manifest is valid, the cmdlet will return true. Otherwise, if the manifest is invalid, the cmdlet will return false and write out a list of validation exceptions.

```
$result = Test-RPSPackageManifest -PatchFilePath c:\package\manifest.xml
```

## Examples

3rd Party Application Package Manifest

This is an example of a manifest used to install Firefox to a Windows Machine:

```xml
<?xml version="1.0" encoding="utf-8"?>
<PackageManifest version="1.0">
  <PackageName>Firefox</PackageName>
  <PackageVersion>1.0.0</PackageVersion>
  <PackageClassification>General</PackageClassification>
  <OsVersion>*</OsVersion>
  <Architecture>x64</Architecture>
  <OsType>Windows</OsType>
  <ProductName>Mozilla Firefox 70.0 (x64 en-US)</ProductName>
  <ProductType>WindowsMsi</ProductType>
  <ProductVersion>70.0</ProductVersion>
  <ProductId>{74994757-3b19-4c54-afe4-ae84e398a3f7}</ProductId>
  <InstallerFileName>firefox.msi</InstallerFileName>
  <SupressReboot>true</SupressReboot>
  <Products />
  <UninstallArguments>/s</UninstallArguments>
  <InstallArguments />
  <Description>This will install Firefox v70 on a All windows x64 bit machines</Description>
  <MsCatalogProductName>Firefox70</MsCatalogProductName>
  <MsCatalogTitle />
  <MsCatalogId />
  <MsCatalogUpdateId />
  <MsCatalogSupercededByKbIds />
  <MsCatalogLinkUrls />
</PackageManifest>
```

### Linux Software Package Manifest

This is an example of a manifest used to install Socat to a Linux Machine:

```xml
<?xml version="1.0" encoding="utf-8"?>
<PackageManifest version="1.0">
  <PackageName>socat</PackageName>
  <PackageVersion>1.7.3.2</PackageVersion>
  <Description>This contains the install for Socat</Description>
  <OsVersion>*</OsVersion>
  <Architecture>x64</Architecture>
  <OsType>Linux</OsType>
  <MsCatalogProductName />
  <MsCatalogTitle />
  <MsCatalogId />
  <Products />
  <MsCatalogUpdateId />
  <PackageClassification>General</PackageClassification>
  <MsCatalogSupercededByKbIds />
  <MsCatalogLinkUrls />
  <UninstallArguments>/s</UninstallArguments>
  <InstallArguments />
  <SupressReboot>true</SupressReboot>
  <ProductName>socat</ProductName>
  <ProductType>LinuxRpm</ProductType>
  <ProductVersion>1.7.3.2</ProductVersion>
  <ProductId>socat</ProductId>
  <InstallerFileName>socat-1.7.3.2-2.el7.x86_64.rpm</InstallerFileName>
</PackageManifest>
```

### Appliance Package Manifest

This package will update an ESX environment:

```xml
<?xml version="1.0" encoding="utf-8"?>
<PackageManifest version="1.0">
  <PackageName>windows8.1-kb4519990-x64</PackageName>
  <PackageVersion>2019.10.8</PackageVersion>
  <Description>This is a test package</Description>
  <OsVersion>*</OsVersion>
  <Architecture>x64</Architecture>
  <OsType>Windows</OsType>
  <MsCatalogProductName>kb4519990</MsCatalogProductName>
  <MsCatalogTitle />
  <MsCatalogId />
  <Products />
  <MsCatalogUpdateId />
  <PackageClassification>General</PackageClassification>
  <MsCatalogSupercededByKbIds />
  <MsCatalogLinkUrls />
  <UninstallArguments>/s</UninstallArguments>
  <InstallArguments />
  <SupressReboot>false</SupressReboot>
  <ProductName>windows8.1-kb4519990-x64</ProductName>
  <ProductType>WindowsHotfix</ProductType>
  <ProductVersion>2019.10.8</ProductVersion>
  <ProductId>kb4519990</ProductId>
  <InstallerFileName>windows8.1-kb4519990-x64.msu</InstallerFileName>
</PackageManifest>
```

# New configurations for CDN

1. DFSRAdmin user account is no longer needed and can be removed from the CMDB *(Optional)*.
2. We no longer need DSC resource Rps_xDFSR.
3. The 'SyncServiceAccount' needs to belong to the 'DFSRAdministrators' group.
4. New Active Directory configurations for 'DFSRAdministrators' group.
    1. On Organizational Unit '**OU CN=Computers,<Domain.Path>**'
        1. AccessControlType = 'Allow'
        2. ActiveDirectoryRights = 'GenericAll'
        3. InheritanceType = 'Descendants'
        4. InheritedObjectType = 'ms-DFSR-LocalSettings'
        5. ObjectType = ''
    2. On Organizational Unit '**OU CN=Computers,<Domain.Path>**'
        1. AccessControlType = 'Allow'
        2. ActiveDirectoryRights = 'CreateChild,DeleteChild'
        3. InheritanceType = 'Descendants'
        4. InheritedObjectType = 'Computer'
        5. ObjectType = 'ms-DFSR-LocalSettings'
5. Nodes need two new properties
    1. ParentCdnProtocol -- How it gets content to and from parent. Bits or DFSR
    2. ChildCdnProtocol -- How it gets or send content between children. Bits or DFSR
6. Need to run Install-RpsTypes PowerShell script.
    1. Located: '<ContentStorePath>\Setup\Configuration\Install-RpsTypes'
7. Need to re-import partial configuration RpsContentDeliveryNetwork.
    1. Located: '<ContentStore>\Dsc\PartialConfigurations'
    2. Get Rps Resource Item
        1. Type = DSCPartial
        2. Name = ContentDeliveryNetwork
    3. $partial.GetResourceAssignments()
        1. Collect current assignments to targets for step **vi**.
    4. Delete Resource Item.
    5. Call 'Import-DscPartial -ContentStore <ContentStorePath>' to re-import.
    6. Reassign partial to targets.

# RPS Package Provider

The RPS Package Provider is a PowerShell module that provides RPS with the ability to Connect to find, download, test, and install patches.

How the Package Provider tests or installs a patch depends on the patch's product type. For more information on the product types see: RPS Patch Product Types

## WindowsHotfix

To **test** if a WindowsHotfix patch is installed the following command is used by the Package Provider:

```
Get-Hotfix -Id <ProductId> -ErrorAction SilentlyContinue
```

- ProductId - The product id from the patch manifest, which is the unique identifier of the product.

To **install** a WindowsHotfix patch the following command is used by the Package Provider:

```
wusa.exe "<Path to installer executable>" /quiet /norestart
```

- Path to installer executable - this is the path to the actual file that kicks off the install process.
  - For example: C:\packages\MyPatch1.0.0\installer.exe

## WindowsMsi

To **test** if a WindowsMsi patch is installed the following command is used by the Package Provider:

```
    $params = @{
        Name            = "<ProductName>"
        RequiredVersion = "<ProductVersion>"
        ProviderName    = 'programs','msi'
        ErrorAction     = 'SilentlyContinue'
    }
    Get-Package @params
```

- ProductName - The product name from the patch manifest that identifies the name of the installed product.
- ProductVersion - The product version from the patch manifest that identifies the version of the installed product.

To **install** a WindowsMsi patch the following command is used by the Package Provider:

```
msiexec.exe /i "<Path to installer msi>" /quiet /norestart
```

- Path to installer msi - this is the path to the actual file that kicks off the install process
  - For example: C:\packages\MyPatch1.0.0\installer.msi

## WindowsMsp

To **test** if a WindowsMsp patch is installed the following command is used by the Package Provider:

```
    $params = @{
        Name            = "<ProductName>"
        RequiredVersion = "<ProductVersion>"
        ProviderName    = 'programs','msi'
        ErrorAction     = 'SilentlyContinue'
    }
    Get-Package @params
```

- ProductName - The product name from the patch manifest that identifies the name of the installed product.
- ProductVersion - The product version from the patch manifest that identifies the version of the installed product.

To **install** a WindowsMsp patch the following command is used by the Package Provider:

```
msiexec.exe /p "<Path to installer msp>" /quiet /norestart
```

- Path to installer msp - this is the path to the actual file that kicks off the install process
  - For example: C:\packages\MyPatch1.0.0\patchInstaller.msp

## WindowsCabinet

> **NOTE**
>
> The RPS Package Provider is only compatible with Windows KB (Knowledge Base) updates that have the CAB file format. Other CAB files may not work using this method. You can test if a CAB file is compatible by using the Dism command explained below to see if its compatible with the RPS Package Provider. If they are not compatible then they may be able to be installed using the script framework and a custom PowerShell script that can install the file.

To **test** if a WindowsCabinet patch is installed the following command is used by the Package Provider:

```
Get-Hotfix -Id <ProductId> -ErrorAction SilentlyContinue
```

- ProductId - The product id from the patch manifest, which is the unique identifier of the product.

To **install** a WindowsCabinet patch the following command is used by the Package Provider:

```
Dism\Add-WindowsPackage -PackagePath <Path to cab archive> -Online -NoRestart
```

- Path to cab archive - this is the path to the actual file for the cab archive
  - for example: C:\packages\MyPatch1.0.0\cabArchive.cab

## WindowsExe

To **test** if a WindowsExe patch is installed use following script which is used internally by the Package Provider:

```
$Name = "<ProductName>"
$IdentifyingNumber = "<ProductId>"
$Version = "<ProductVersion>"

$uninstallRegistryKey = 'HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall'
$uninstallRegistryKeyWow64 = 'HKLM:\SOFTWARE\Wow6432Node\Microsoft\Windows\CurrentVersion\Uninstall'

$productEntry = $null
$productEntryPath = $null

if ([String]::IsNullOrEmpty($Version) -or ([String]::IsNullOrEmpty($IdentifyingNumber) -and
[String]::IsNullOrEmpty($Name)))
{
    return $productEntry
}

if (-not [String]::IsNullOrEmpty($IdentifyingNumber))
{
    $productEntryKeyLocation = Join-Path -Path $uninstallRegistryKey -ChildPath $IdentifyingNumber
    $productEntryPath = Get-Item -Path $productEntryKeyLocation -ErrorAction 'SilentlyContinue'

    if ($null -eq $productEntryPath)
    {
        $productEntryKeyLocation = Join-Path -Path $uninstallRegistryKeyWow64 -ChildPath $IdentifyingNumber
        $productEntryPath = Get-Item -Path $productEntryKeyLocation -ErrorAction 'SilentlyContinue'
    }

    if ($productEntryPath -and $Version -eq (Get-LocalizedRegistryKeyValue -RegistryKey $productEntryPath -
ValueName 'DisplayVersion'))
    {
        Write-Host "True"
    }
}
else
{
    foreach ($registryKeyEntry in (Get-ChildItem -Path @( $uninstallRegistryKey, $uninstallRegistryKeyWow64) -
ErrorAction 'Ignore' ))
    {
        if ($Name -eq (Get-LocalizedRegistryKeyValue -RegistryKey $registryKeyEntry -ValueName 'DisplayName')
-and
            $Version -eq (Get-LocalizedRegistryKeyValue -RegistryKey $registryKeyEntry -ValueName
'DisplayVersion'))
        {
            Write-Host "True"
        }
    }
}

Write-Host "False"
```

- ProductName - The product name from the patch manifest that identifies the name of the installed product.
- ProductVersion - The product version from the patch manifest that identifies the version of the installed product.
- ProductId - The product id from the patch manifest, which is the unique identifier of the product.

To **install** a WindowsExe patch the following command is used by the Package Provider:

```
<Path to executable> <InstallArguments>
```

- Path to executable - this is the path to the actual file for the exe
  - For example: C:\packages\MyPatch1.0.0\installer.exe /quiet

## LinuxRpm

To **test** if a LinuxRpm patch is installed the following command is used by the Package Provider:

```
rpm -qa <ProductName> | grep <ProductVersion>
```

- ProductName - The product name from the patch manifest that identifies the name of the installed product.
- ProductVersion - The product version from the patch manifest that identifies the version of the installed product.

To **install** a LinuxRpm patch the following command is used by the Package Provider:

```
yum -y --nogpgcheck localinstall <Path to RPM>
```

- Path to RPM - this is the path to the actual file for the RPM
  - For example: /root/packages/MyPatch1.0.0/installer.rpm

## ScriptFramework

To **test** if a ScriptFramework patch is installed you can run the "Test" method of your ScriptFramework script directly.

To **install** a ScriptFramework patch you can execute the "Set" method of your ScriptFramework script directly.

For more information and examples on this see: Packaging Script Framework

# How to Patch using RPS

This document describes the Patching process of the Rapid Provisioning System (RPS).

## Prerequisites

For the Content Delivery Network to transfer content between nodes will will need to know which protocol to use between nodes. (Bits or DFSR). To configure this setting, on each local node we will need to create two new properties.

- ParentCdnProtocol
  - The value will be either 'Dfsr' or 'Bits'. This is the connection Rps will use when communicating with its parent.

- ChildCdnProtocol
  - The value will be either 'Dfsr' or 'Bits'. This is the connection Rps will use when communicating with all of it's children. You can not configure each child differently.

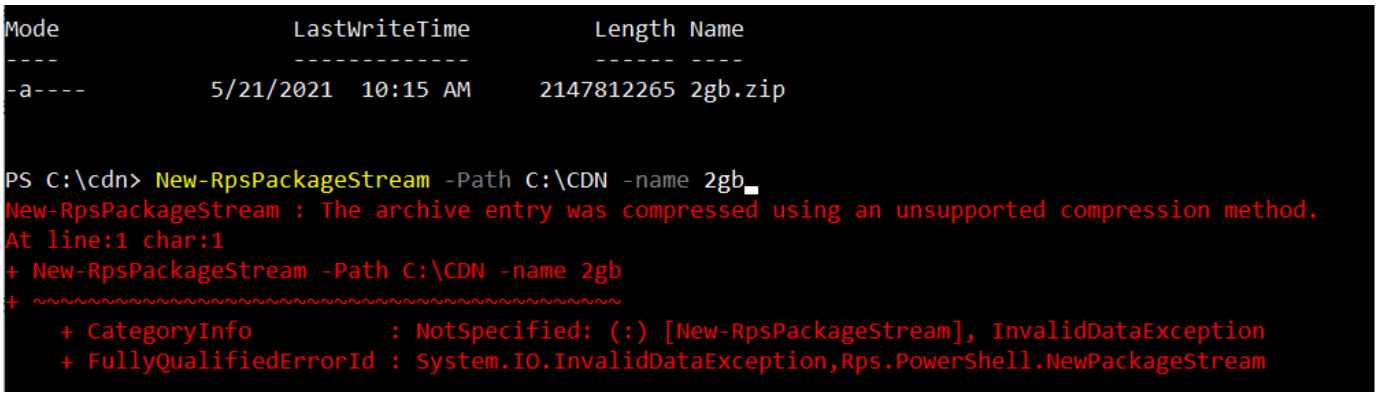
## Known Limitations

The maximum supported RPS Patch size is 2GB. Any RPS Patch zip file that is larger than 2GB will throw an exception when RPS tries to open the patch and read the manifest file from the patch zip file.

- This exception can occur in two scenarios:

  - When creating a new Package Stream with a package where the zip file size is greater than 2GB.
  - When adding a new patch to an existing Package Stream where the patch zip file size is greater than 2GB.

  **The resulting behavior:**



```
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         5/21/2021  10:15 AM     2147812265 2gb.zip


PS C:\cdn> New-RpsPackageStream -Path C:\CDN -name 2gb
New-RpsPackageStream : The archive entry was compressed using an unsupported compression method.
At line:1 char:1
+ New-RpsPackageStream -Path C:\CDN -name 2gb
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [New-RpsPackageStream], InvalidDataException
    + FullyQualifiedErrorId : System.IO.InvalidDataException,Rps.PowerShell.NewPackageStream
```

# How to Patch using RPS

RPS Patching feature all users to deploy patches and new software through the RPS system. This section will provide the information on supported patch types, support operating systems, and instructions on using RPS to patch.

## Supported Patch Types:

The following are a list of valid patch types that can be applied using RPS:

- MSU (Windows)

- EXE (Windows)

- MSI (Windows)

- CAB (Windows)

- RPM (Linux)
- Script Framework (PowerShell)

## Supported Operating Systems:

RPS supports the following operating systems and versions:

- Windows 10
- Windows Server 2012 (or newer)
- CentOS Linux 5 (or newer)
- Red Hat Linux 5 (or newer)
- Appliances using PowerShell.

## How to Patch Windows or Linux

> **NOTE**
>
> For a Package Stream to deploy the target(s) must be in a valid maintenance window.

1. (optional) Create Maintenance Window(s) for the Targets that you want to patch (see external document for instructions on how to create a Maintenance Window)
2. Create the Package file
    1. Create the Package.RPS manifest (see external document for instructions on how to create a Package Manifest)
        1. The manifest will contain information that describes the patch, including supported OsType, OsVersion, and Architecture which will be used when assignments are made automatically between the Packages and Targets.
    2. Create a zip file that contains the Package.RPS, the installer file (e.g. msu, exe, msi, rpm, cab, Powershell, etc.), and any other supporting files needed by the installer.
        1. The manifest and the main installer file should in the root of the zip archive
3. Place all the Package files in a single directory (there should be a single directory for each Package Stream)
4. Copy the Package files to the Node. You can use an external media device to do this, such as a thumb drive to copy the files to the Node.
5. Create the Package Stream for the Package files (see external document for instructions on how to create a Package Stream)
6. Approve the Package Stream (see external document for instructions on how to approve a Package Stream)

## How to Patch an Appliance

1. Create a Target type that is 'patchable' and properties for the appliance. see: How to Create a Patchable Target Type
2. Create the Package zip file that contains:
    1. The actual patch installer/executable for the appliance
    2. The manifest. Create this by following the external documentation on how to create a package manifest.
        1. The manifest for the appliance patch needs to have the following information:

        1. OsType - ESX (this should match the OsType of the Target Item property)
        2. Architecture - * (this can be a wildcard to match any architecture or a specific architecture such as x86)
    3. The PowerShell script that will perform the work to run the installer/executable to patch the appliance. This can be any script as needed but if it has parameters then you can add these as additional properties to the Target Item that was created in step #3. For more information on the Script Framework see: Packaging Script Framework
3. Place the Package file(s) in a single directory (there should be a single directory for each Package Stream)
4. Copy the Package file(s) to the Node. You can use an external media device to do this, such as a thumb drive to copy the files to the Node.
5. Create the Package Stream for the Package file(s) (see external document for instructions on how to create a Package Stream)
6. Approve the Package Stream (see external document for instructions on how to approve a Package Stream)

# How to create a patchable Target Type

To create a target type and properties for a patchable target be sure to use the -CanPatch switch parameter to mark it as being patchable.

A patchable target type can be used to create target items that are patchable. Only target items that are created with a target type that has the CanPatch attribute are able to be patched.

The following example lists the **minimum** properties required to create a patchable target type:

```
$hostType = Set-RpsTargetType -Name 'EsxHost' -IsRoot -CanPatch
$null = Set-RpsTypeProperty -Parent $hostType -Name 'ComputerName' -PropertyType Text -IsRequired
$null = Set-RpsTypeProperty -Parent $hostType -Name 'IPAddress' -PropertyType Text -IsRequired
$null = Set-RpsTypeProperty -Parent $hostType -Name 'RunPackagesOnSma' -PropertyType Boolean -DefaultValue
'True'
$null = Set-RpsTypeProperty -Parent $hostType -Name 'OsType' -PropertyType Text - DefaultValue 'ESX'
$null = Set-RpsTypeProperty -Parent $hostType -Name 'Architecture' -PropertyType Text - DefaultValue 'x86'
$null = Set-RpsTypeProperty -Parent $hostType -Name 'OsVersion' -PropertyType Text - DefaultValue '10.12.10'
```

Description of the type properties:

```
| Property Name    | Property Type | Description   |
| ---------------- |:--------------:| ------------:|
| ComputerName     | text          | Name of the Target |
| IPAddress        | text          | IP Address of the Target |
| RunPackagesOnSma | boolean       | Indicates whether to deploy Packages from SMA instead of from the target
itself (SMA acts as a proxy) |
| OsType           | text          | OS Type (e.g. Linux, Windows, ESX) |
| Architecture     | text          | Architecture of the Target |
| OsVersion        | text          | OS Version |
```

# How to use Maintenance Windows

## Maintenance Windows

If software should only be installed for a given endpoint in RPS during one or more specific times, then Maintenance Windows could help.

## What is a Maintenance Window

The Maintenance Window in RPS is a *templatized* Resource Item that gets assigned to one or more Target Items (computer endpoints) and represents a window of time that software installation can occur on those Target Items.

This Resource Item is additive in nature (meaning multiple Maintenance Windows can be specified). An endpoint can have multiple Maintenance Windows. Maintenance Windows are particularly useful to schedule around a period of no change (PON) or code freeze.

> **Tip:** Without an active Maintenance Window, no Packages will be applied or removed because the Desired State Configuration Set command will only run inside a Maintenance Window.

# Using Maintenance Windows

## Pre-requisites

- RPS Installation
- A host in the network with the RPS Website installed
- The RPS Website and navigate to the Packaging Page:

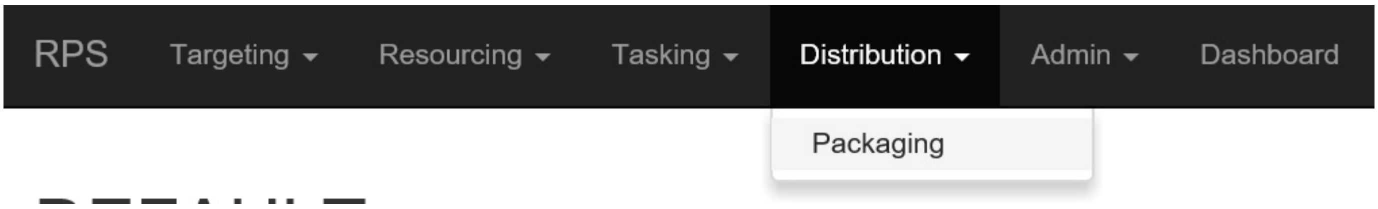


Figure 1: Package Management Page

- Select the "Schedule" tab from the Packaging Page:

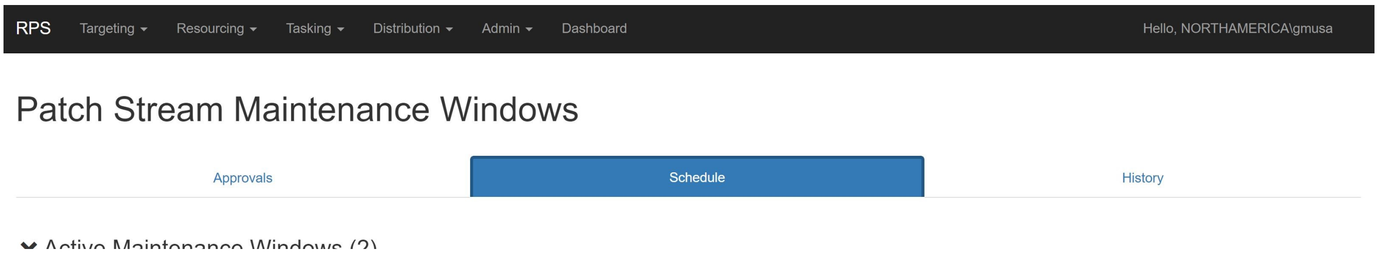

RPS
Targeting
Resourcing
Tasking
Distribution
Admin
Dashboard
Hello, NORTHAMERICA\gmusa
Patch Stream Maintenance Windows
Approvals
Schedule
History
Active Maintenance Windows (2)

Figure 2: Schedule Tab

During the installation process for RPS (e.g. executing `Install-Rps.ps1`) the prerequisites (such as Template Type Definitions) and the RPS Website are installed and configured on at least one Virtual Machine. From that machine, the RPS Website is hosted to facilitate configuring Maintenance Windows for software installation using Desired State Configuration (DSC).

**NOTE**

The Desired State Configuration (DSC) commands for **Get** and **Test** will always run and return results. **Set**, however, will only run inside a

Maintenance Window for that endpoint.

## Create Maintenance Windows

There are three mandatory properties of a Maintenance Window: `Name`, `Frequency`, and `StartDate`.

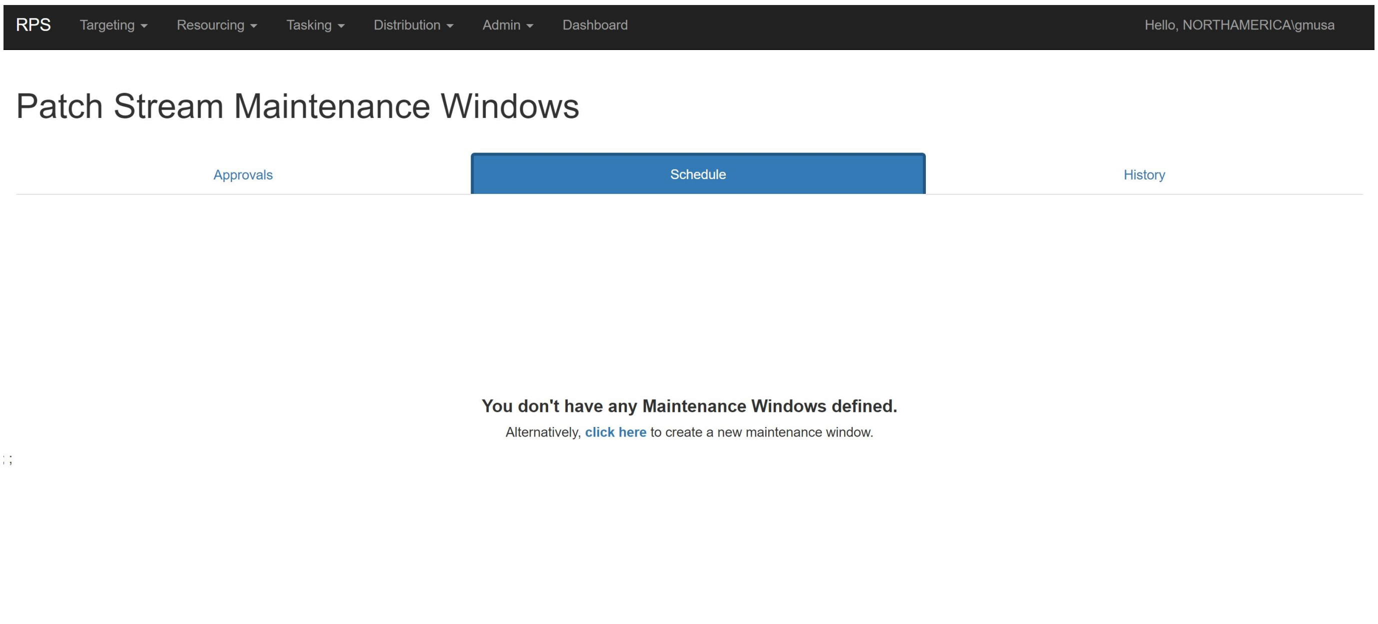
You can see all of the recurrence related properties and their data types at Maintenance Window Recurrence Patterns.

### How to Create a Maintenance Window from the UI

1. As outlined in the introduction, launch the RPS Website, navigate to the Package Page, and select the "Schedule" tab.

2. Click the "Create New Maintenance Window" button

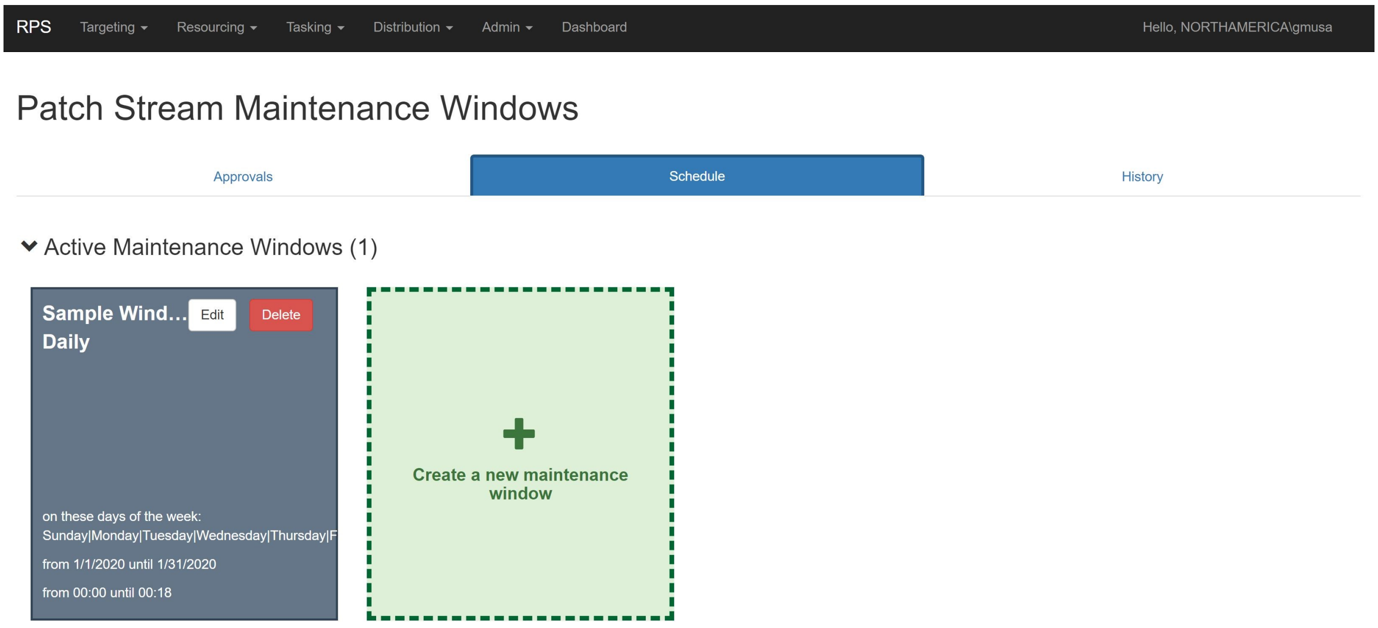
   If there are no existing Maintenance Windows, you'll be prompted to create one from the screen that appears:



You don't have any Maintenance Windows defined.
Alternatively, click here to create a new maintenance window.

   Figure 3: No Existing Maintenance Window

   If there are Maintenance Windows already configured, you'll be able to create another one using the green tile:


RPS
Targeting
Resourcing
Tasking
Distribution
Admin
Dashboard
Hello, NORTHAMERICA\gmusa
Patch Stream Maintenance Windows
Approvals
Schedule
History
Active Maintenance Windows (1)
Sample Wind...
Edit
Delete
Daily
Create a new maintenance window

on these days of the week:
Sunday|Monday|Tuesday|Wednesday|Thursday|F
from 1/1/2020 until 1/31/2020
from 00:00 until 00:18

   Figure 4: Green Create Window Tab

3. Fill in the required form fields

See Maintenance Window Recurrence Patterns for details on how to create a valid recurrence pattern.

4. Optionally specify which Target Items the Maintenance Window should apply to

   See Specifying Target Items for details on how to make Target Item selection.

5. Save the Maintenance Window by clicking the "Submit" button

## Create a Maintenance Window from PowerShell

Here's how to create a new Maintenance Window from PowerShell:

1. Create the Maintenance Window using the `New-RpsMaintenanceWindow` command and specify the three required properties `Name`, `Frequency`, and `StartDate`

2. Optionally, specify additional properties by property name:

```
    $startDate = (Get-Date)
    $startTime = New-TimeSpan -Hours 0 -Minutes 0
    $endDate = $startDate.AddDays(5)
    $endTime = New-TimeSpan -Hours 11 -Minutes 59
    $frequency = "Daily"

    $myWindow = New-RpsMaintenanceWindow -Name 'myWindow' -Frequency $frequency -StartDate $startDate -
StartTime $startTime -EndDate $endDate -EndTime $endTime
```

3. Checking the result of `$myWindow` will show you what was created, for example here's `$myWindow.Properties`:

| KEY | VALUE |
|---|---|
| StartDate | 1/30/2020 12:00:00AM |
| StartTime | 00:00 |
| EndTime | 23:59 |
| EndDate | 2/4/2020 12:00:00AM |
| Frequency | "Daily" |

**NOTE**

See Maintenance Window Recurrence Patterns for the data types to pass for different Maintenance Window properties.

## Read Maintenance Windows

### Viewing Maintenance Windows from the UI

1. As outlined in the introduction, launch the RPS Website, navigate to the Package Page, and select the "Schedule" tab.

### Viewing Maintenance Windows from PowerShell

Existing Maintenance Windows can be viewed and stored into memory via the `Get-RpsMaintenanceWindow` command.

1. You can retrieve a Maintenance Window by Name or Id like so:

```
   Get-RpsMaintenanceWindow -Name 'myWindow'
```

or

```
Get-RpsMaintenanceWindow -Id <GUID>
```

## Update Maintenance Windows

### Update Existing Maintenance Window from the UI

Existing Maintenance Windows can be updated from the Schedule tab of the Patch Stream Maintenance page.

1. As outlined in the introduction, launch the RPS Website, navigate to the Package Page, and select the "Schedule" tab.
2. Click the "Edit" button on the top-right of the maintenance window tile (outlined here in yellow).

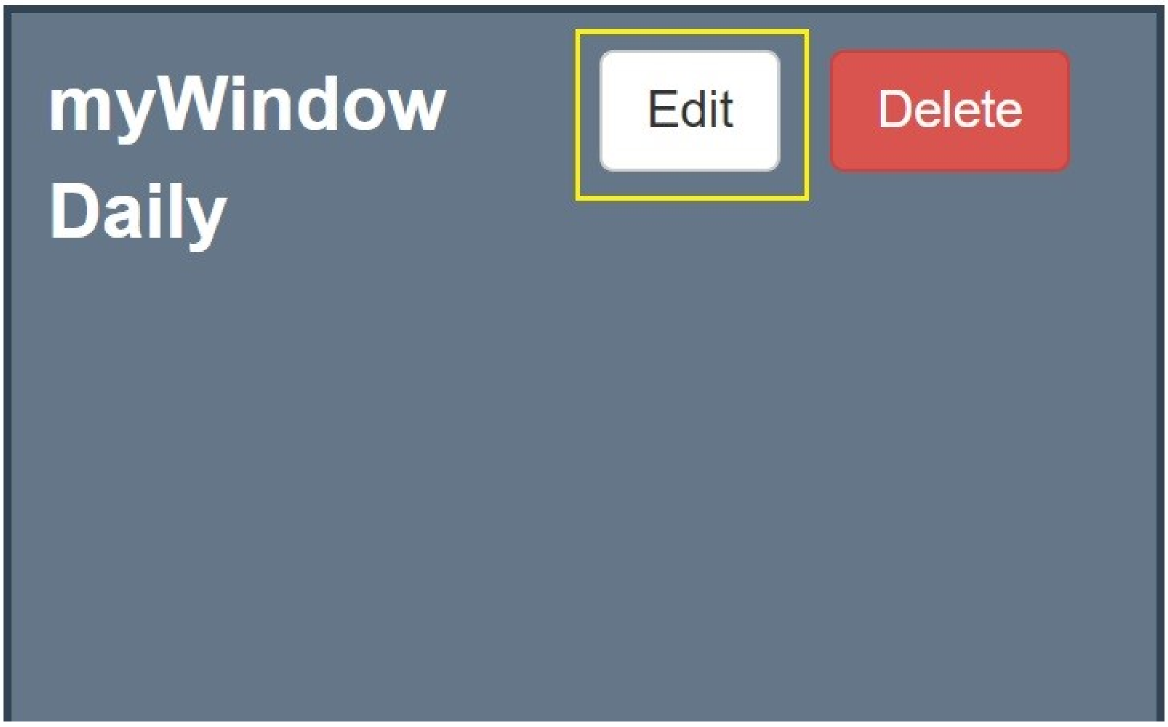


Figure 9: Edit Button

3. Make the desired changes. Click "Submit".

### Update Existing Maintenance Window from PowerShell

Here's how to update existing Maintenance Windows from PowerShell. For this example, we'll inspect the Maintenance Window's current properties with `Get-RpsMaintenanceWindow` and then decide to update it.

You can, however, directly update the Maintenance Window by executing `Set-RpsMaintenanceWindow`:

1. We'll get the Maintenance Window using `Get-RpsMaintenanceWindow`:

```
$myWindow = Get-RpsMaintenanceWindow -Name 'myWindow'
```

2. And we decide to change the Day Of Week the window applies to by passing the updated values as parameters to the `Set-RpsMaintenanceWindow` command

```
Set-RpsMaintenanceWindow -Id $myWindow.Id -DayOfTheWeek 'Sunday'
```

or

```
Set-RpsMaintenanceWindow -Name 'myWindow' -DayOfTheWeek 'Sunday'
```

## Delete Maintenance Windows

### Delete a Maintenance Window from the UI

Maintenance Windows can be deleted from the UI in the Schedule Tab from the Patch Stream Management page.

1. As outlined in the introduction, launch the RPS Website, navigate to the Package Page, and select the "Schedule" tab.
2. Click the "Delete" button on the top-right of the maintenance window you want to delete.

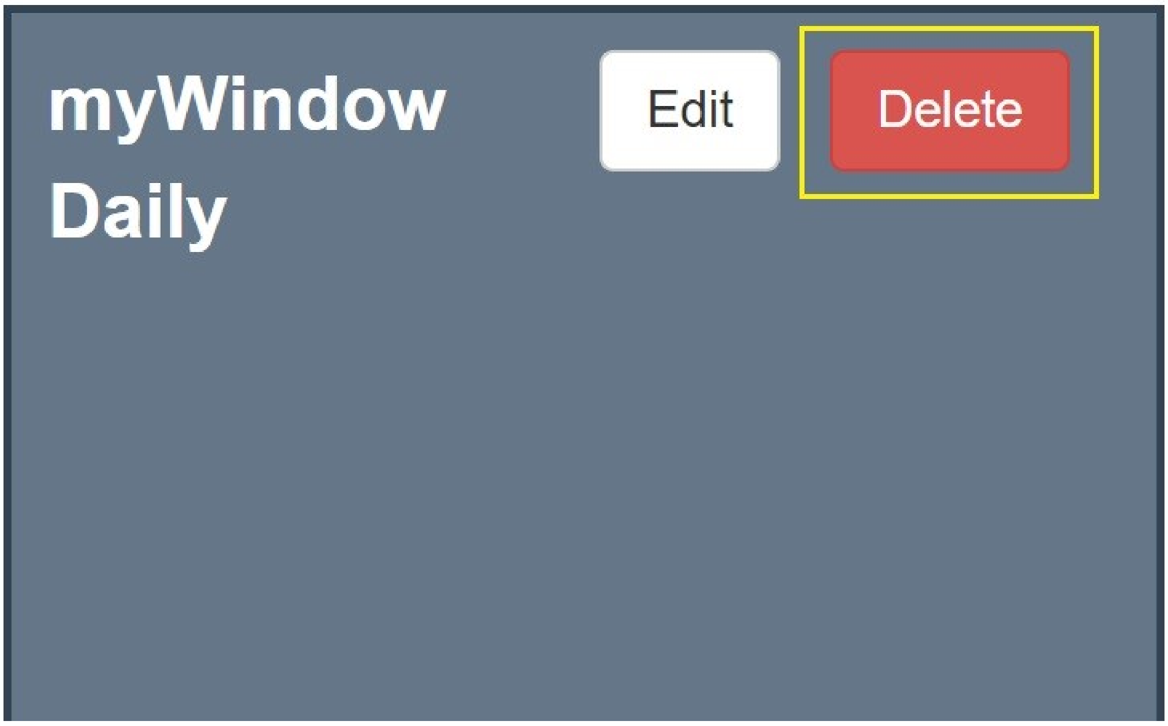
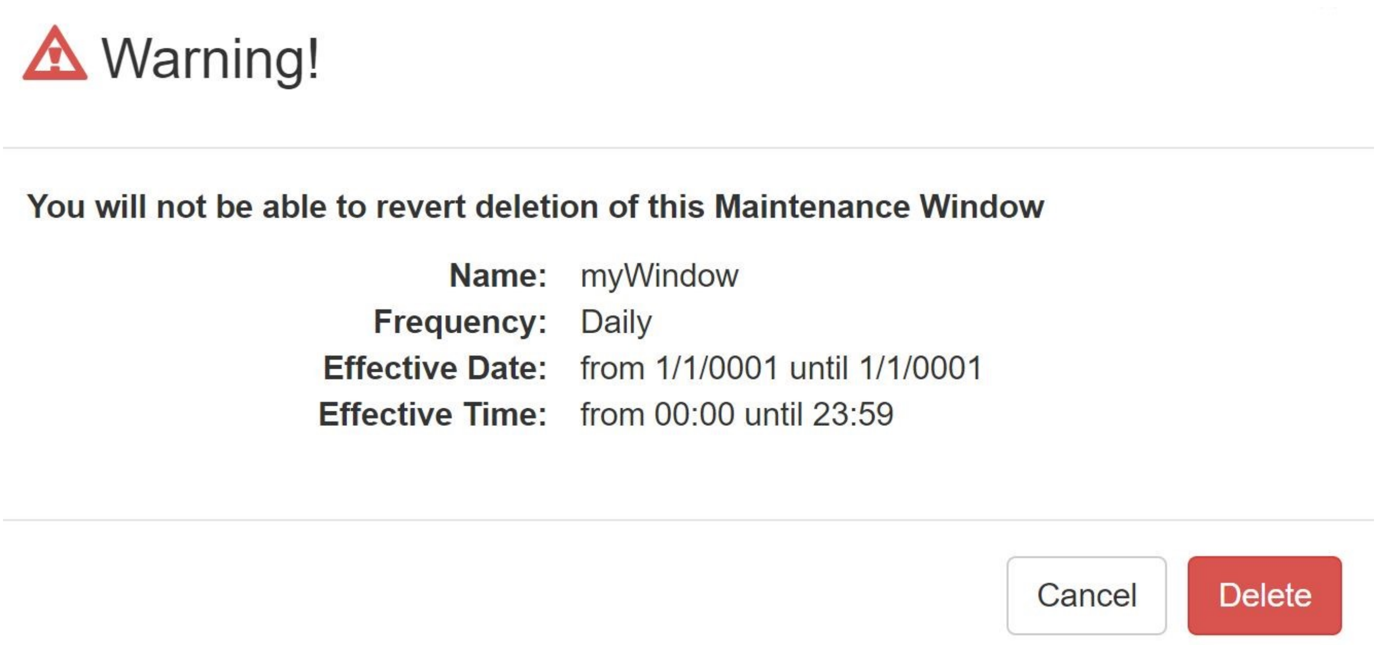


Figure 13: Delete Button

3. A confirmation dialog appears that informs this action is permanent and some metadata is displayed to confirm this is the correct window to delete.



Warning!

You will not be able to revert deletion of this Maintenance Window

|  |  |
|---|---|
| Name: | myWindow |
| Frequency: | Daily |
| Effective Date: | from 1/1/0001 until 1/1/0001 |
| Effective Time: | from 00:00 until 23:59 |

Figure 14: Warning Dialog Box

### Delete a Maintenance Window from PowerShell

Here's how to delete a Maintenance Window from PowerShell:

1. Delete the Maintenance Window using the `Delete-RpsMaintenanceWindow` command:

```
Delete-RpsMaintenanceWindow -Id <GUID>
```

## Maintenance Window Recurrence Patterns

**NOTE**

Not all fields should be populated for a single Maintenance Window. Instead, create multiple Maintenance Windows, as needed, to satisfy the time/date/frequency requirements.

| PROPERTY | INTERPRETED TYPE | DESCRIPTION |
|---|---|---|
| Name | string | The friendly name of the Maintenance Window. { e.g. 'Patch Tuesday' } |
| Frequency | string | The frequency upon which the window should be open { Daily, Weekly, or Monthly } |
| StartDate | DateTime | The first UTC date Set-TargetResource is allowed to run |
| EndDate | DateTime | The last UTC date Set-TargetResource is allowed to run |
| StartTime | TimeSpan | The first time the of day Set-TargetResource is allowed to run |
| EndTime | TimeSpan | The last time of day the Set-TargetResource is allowed to run |
| Day | int[] | The day(s) of the month Set-TargetResource will run. 0 represents the last day of the month. { 0 - 31 } |
| DayOfWeek | string[] | The day(s) of the week Set-TargetResource will run. { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, or Saturday } |
| Week | int[] | The week(s) of the month Set-TargetResource could be run. 0 represents the last week of the month. { 0, 1, 2, 3, or 4 } |

**NOTE**

A value of **0** for Day or Week indicate the last day or week of the month, respectively.

**NOTE**

Specify UTC dates `StartDate` and `EndDate` so that packages are applied in a predictable timezone regardless of target item timezone.

## Specifying Target Items

Maintenance Windows can apply to zero to many Target Items through use of ResourceAssignments.

### Specifying Target Items from the UI

In the UI, these Resource Assignments are made using a selectable tree view. Items with checkmarks next to them will be assigned, items without checkmarks will not be assigned.

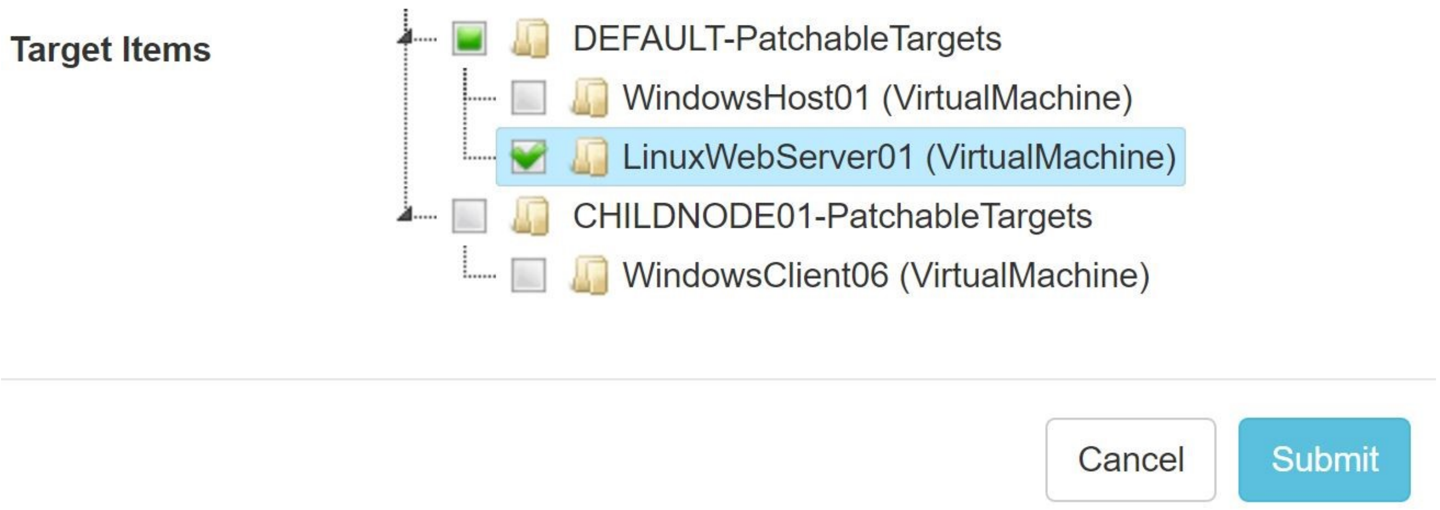

Target Items
DEFAULT-PatchableTargets
WindowsHost01 (VirtualMachine)
LinuxWebServer01 (VirtualMachine)
CHILDNODE01-PatchableTargets
WindowsClient06 (VirtualMachine)
Cancel
Submit

Figure 15: Selectable Tree View

1. Navigate to the Schedule tab of the Patch Stream Management page.
2. Create a Maintenance Window (or update an existing one using the Edit button)
3. Select one or many Target Items to assign this Maintenance Window to using the tree view that appears. Checking a TargetGroup will select/de-select all of the TargetItems in that group.

### Specifying Target Items from PowerShell

In Powershell, we specify which Target Items a Maintenance Window applies to by making Resource Assignments:

1. Fetch the Maintenance Window to assign Target Items to

```
$myWindow = Get-RpsMaintenanceWindow -Name 'myWindow'
```

2. Fetch the Target Item(s) to assign the Maintenance Window to

```
$myTargetItem = Get-RpsTargetItem -Name 'myTargetItem'
```

3. Create a Resource Assignment with those items:

```
New-RpsResourceAssignment -ResourceItem $myWindow.MaintenanceWindowResourceItem.Id -TargetItem
$myTargetItem
```

# How To Create A RPS Package

## What is a Package

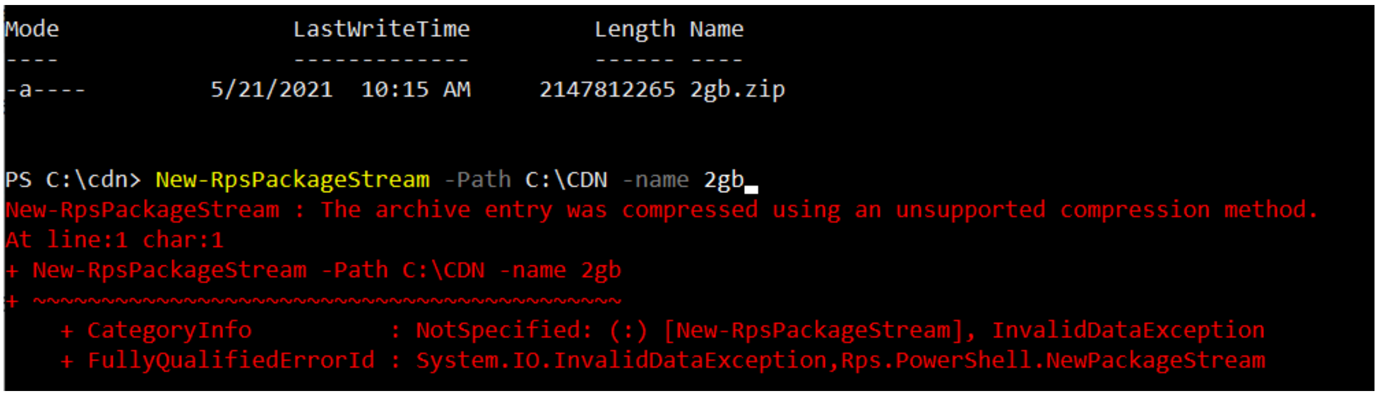
A package is what RPS calls any executable meant to be distributed via RPS to other RPS Target with the intent of managing the software on the target. A package file is a Zip archive that contains the content needed in order to manage the software as well as a Package Manifest file. That zip archive in its entirety is what would be considered a "Package" in terms of RPS, not the individual files inside.

## Known Limitations

The maximum supported RPS Package size is 2GB. Any RPS Package zip file that is larger than 2GB will throw an exception when RPS tries to open the package and read the manifest file from the package zip file.

- This exception can occur in two scenarios:
  - When creating a new package stream with a package where the zip file size is greater than 2GB.
  - When adding a new package to an existing package stream where the package zip file size is greater than 2GB.

  **The resulting behavior:**



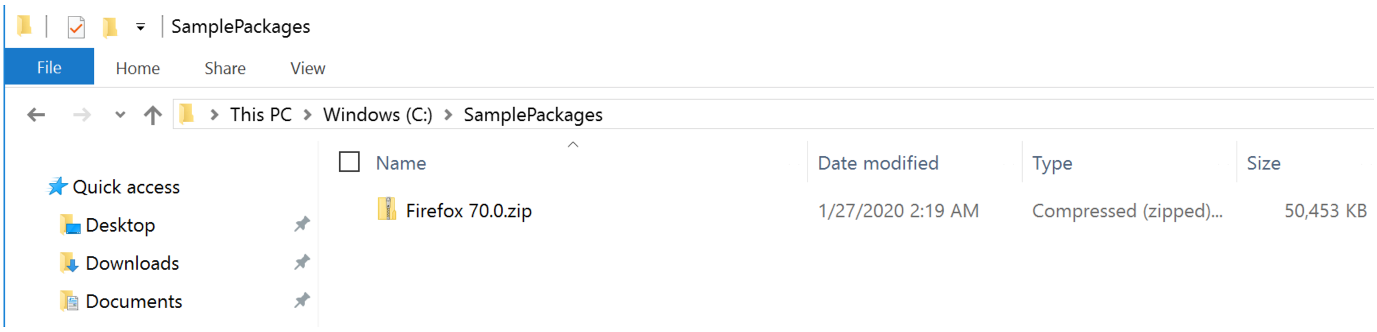
```
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         5/21/2021  10:15 AM     2147812265 2gb.zip


PS C:\cdn> New-RpsPackageStream -Path C:\CDN -name 2gb
New-RpsPackageStream : The archive entry was compressed using an unsupported compression method.
At line:1 char:1
+ New-RpsPackageStream -Path C:\CDN -name 2gb
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [New-RpsPackageStream], InvalidDataException
    + FullyQualifiedErrorId : System.IO.InvalidDataException,Rps.PowerShell.NewPackageStream
```

## How to Create a Package

A package consists of all the files needed to run on the targets in order to install, upgrade, or uninstall. A package archive requires the content, of which one file should be the executable entry point, and a Package Manifest file, which contains the metadata needed by RPS in order to process the package.

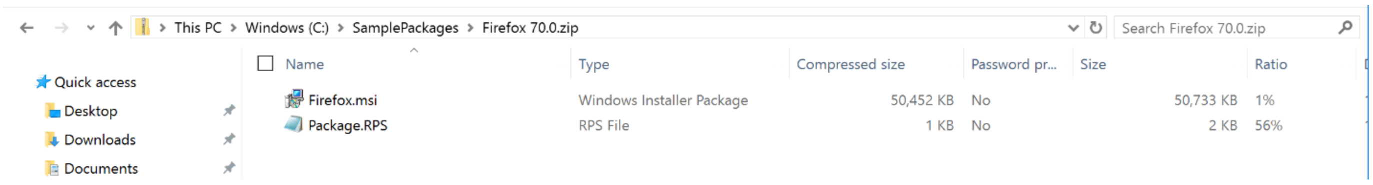
Example: Creating a FireFox Install Package

1. Zip the installer and any files needed for the desired Firefox version.
2. Create the Package Manifest file with a name of "Package.RPS" for the Firefox version that is to be installed
3. Add the Package Manifest to the Firefox zip archive created above



| Name | Date modified | Type | Size |
|---|---|---|---|
| Firefox 70.0.zip | 1/27/2020 2:19 AM | Compressed (zipped)... | 50,453 KB |

Figure 1 Firefox v70.0 Rps Package



This PC > Windows (C:) > SamplePackages > Firefox 70.0.zip

| Name | Type | Compressed size | Password pr... | Size | Ratio |
|---|---|---|---|---|---|
| Firefox.msi | Windows Installer Package | 50,452 KB | No | 50,733 KB | 1% |
| Package.RPS | RPS File | 1 KB | No | 2 KB | 56% |

Figure 2 Firefox Package Contents

# Non Patching Content Delivery

Rps can transfer other types of content other than patching. i.e. Log shipping.

## Type Definition

Creating transferable content starts with the type definition. The resource type definition tells Rps to include this type when it configures DFSR and Bits.

```
$type = Set-RpsResourceType -Name 'Logs' -IsContentDistribution -CdnDirection Downstream
$null = Set-RpsTypeProperty -Parent $type -Name 'DisplayName' -PropertyType Text
```

This command will configure any Resource Item of type 'Logs' to be treated as content. It will also configure the delivery path as Downstream, the other option is Upstream.

## Create a Resource Item

Now we will create a Resource Item that will give us a new directory in the CDN folder for us to add our content.

```
$newContent = New-RpsResourceItem -Type 'Logs' -Name 'Firewall Logs' -IsGlobal $true -Properties @{
DisplayName = 'Firewall Logs' }

# Get the folder Id. FolderId is the reverse of the content resource item
$folderId = [Rps.Api.Utils.HashingUtils]::Reverse($newContent.Id)

# Set FolderResourceId on the new content.
$newContent.FolderResourceId = $folderId
$newContent.Update()

# Get Cdn Folder
$localNode = Get-RpsLocalNode
$cdnPath = $localNode.CdnPath

# Create folder
$null = New-Item -Path $cdnPath -Name $folderId -ItemType Directory

# Configure Folder data in CMDB
$fileCatalog = [Rps.Api.Utils.FileUtils]::BuildFolderCatalog($(Join-Path -Path $cdnPath -ChildPath $folderId))
[Rps.Api.Utils.FileUtils]::CreateFolderCatalog($fileCatalog, $contentName, $folderId)
```

This code will create a new ResourceItem with type Logs, and will set the FolderResourceId on the resource. The Folders Resource Item id is the reverse guid of the ResourceItem. Then we create a new folder in the CDN for us to add our content to and make a FolderCatalog which will add additional required data in the CMDB.

## Assign Resource Item to targets

For the content to be transferred to other nodes we will need to assign them to a target on that node. Since the parent node must have all content its children need we can assign this resource item to a target on the lowest node.

```
$siteTarget = Get-RpsTargetItem -Name app-s.region.rps -Type VirtualMachine
$null = $newContent.AssignTo($siteTarget)
```

Example

```powershell
$contentType = 'Logs'
$contentName = 'FirewallLogs'

# Create Type Definition for new Content
$type = Set-RpsResourceType -Name 'Logs' -IsContentDistribution -CdnDirection Downstream
$null = Set-RpsTypeProperty -Parent $type -Name 'DisplayName' -PropertyType Text

# Create the resource item with the same type
$newContent = New-RpsResourceItem -Type $contentType -Name $contentName -IsGlobal $true -Properties @{
DisplayName = 'Firewall Logs' }

# Get the folder Id. FolderId is the reverse of the content resource item
$folderId = [Rps.Api.Utils.HashingUtils]::Reverse($newContent.Id)

# Set FolderResourceId on the new content.
$newContent.FolderResourceId = $folderId
$newContent.Update()

# Get Cdn Folder
$localNode = Get-RpsLocalNode
$cdnPath = $localNode.CdnPath

# Create folder
$null = New-Item -Path $cdnPath -Name $folderId -ItemType Directory

# Configure Folder data in CMDB
$fileCatalog = [Rps.Api.Utils.FileUtils]::BuildFolderCatalog($(Join-Path -Path $cdnPath -ChildPath $folderId))
[Rps.Api.Utils.FileUtils]::CreateFolderCatalog($fileCatalog, $contentName, $folderId)

# Assign content to targets on each node you want to send the data to
$regionTarget = Get-RpsTargetItem -Name app.region.rps -Type VirtualMachine
$siteTarget = Get-RpsTargetItem -Name app-s.region.rps -Type VirtualMachine

$null = $newContent.AssignTo($regionTarget)
$null = $newContent.AssignTo($siteTarget)
```

# How to load a Package Stream

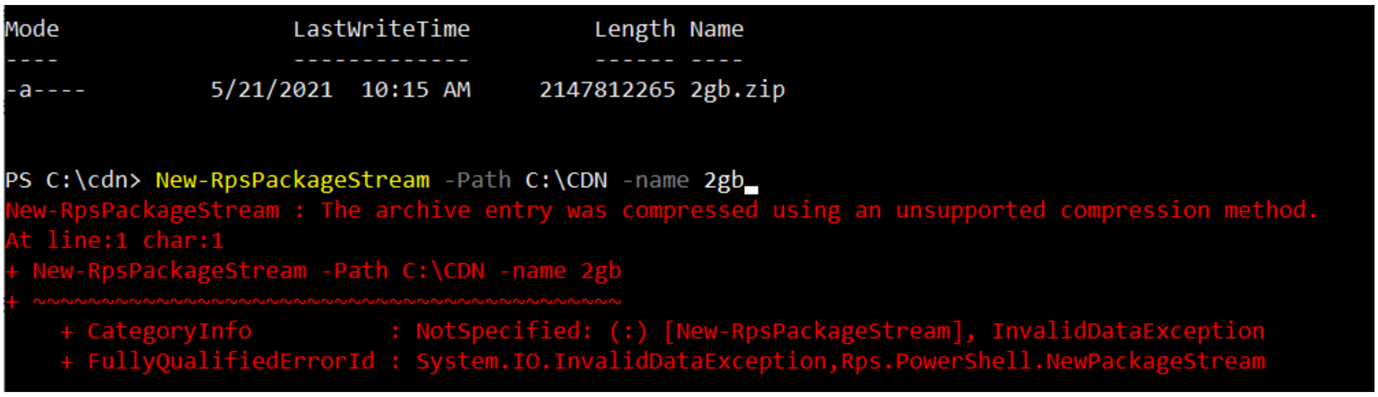
This document describes the step by step instructions from end to end for loading a Package Stream.

## Known Limitations

The maximum supported RPS Package size is 2GB. Any RPS Package zip file that is larger than 2GB will throw an exception when RPS tries to open the Package and read the manifest file from the Package zip file.

- This exception can occur in two scenarios:
    - When creating a new Package Stream with a Package where the zip file size is greater than 2GB.
    - When adding a new Package to an existing Package Stream where the Package zip file size is greater than 2GB.

  **The resulting behavior:**



```
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         5/21/2021  10:15 AM     2147812265 2gb.zip


PS C:\cdn> New-RpsPackageStream -Path C:\CDN -name 2gb
New-RpsPackageStream : The archive entry was compressed using an unsupported compression method.
At line:1 char:1
+ New-RpsPackageStream -Path C:\CDN -name 2gb
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [New-RpsPackageStream], InvalidDataException
    + FullyQualifiedErrorId : System.IO.InvalidDataException,Rps.PowerShell.NewPackageStream
```

## How to create a Package Stream using PowerShell cmdlets

**NOTE**

Steps 1 and 2 can be skipped if you do not need to copy Packages to another node.

1. Gather the Package zip files and place the packages on an external media device (e.g. USB thumb drive) for loading on to the desired node.

2. Place all of the Package zip files into a single directory on the external media device. For example: E:\Packages\

3. Log in to the node where you will be creating the Package Stream with a user account that has permission to the CDN folder. The Active Directory group that the user should belong to is: ContentCreators. This group will grant permission to add Packages to the CDN.

4. Open a PowerShell window

5. Change your directory to the RPS ContentStore. For example:

   ```
   cd C:\ContentStore
   ```

6. Import the RPS API module

   ```
   Import-Module C:\ContentStore\Modules\Rps-Api
   ```

7. Create the Package stream using the New-RpsPackageStream cmdlet.

   Options for the New-RpsPackageStream cmdlet are:

| PARAMETER NAME | TYPE | DESCRIPTION |
|---|---|---|
| Name | string | Name of the Package Stream (max length = 255) |
| Path | string | Path of the Packages |
| PackageExtensions | List of String | List of file extensions that will be used when searching for Packages in the Path. The files can have any extension but must be valid ZIP archives. |
| Recurse | Switch | Indicates whether to search all sub-directories for Packages. |

Example of the New-RpsPackageStream cmdlet. The following example will create a Package Stream named "MyPackageStream1" that contains Packages in the C:\Packages\ directory.

```
New-RpsPackageStream -Name MyPackageStream1 -Path C:\Packages -PackageExtenstions ".zip"
```

8. After the Package Stream command has been executed the following will happen:

a. A Package Stream item will be added to the Content Management Database (CMDB).

b. The Package items will be added to the CMDB, one for each Package found in the provided "Path" of the Packages.

c. The Packages will be copied in to the CDN directory, so they are able to replicate across the CDN.

**NOTE**

The initial state of a Package Stream is "Pending" and therefore no assignments between Packages and Targets will be made. Once a Package Stream's status is "Approved" then assignments will be created between the Packages and applicable Targets.

An assignment between a Package and a Target means that the following match between the two: OS Type (e.g. Linux, Windows), OS Version (e.g. 10, 8.1, 2012 R2), and architecture (e.g. x86, x64, ARM, etc.). If an assignment had been made then the Package is able to be deployed to the assigned target(s).

A target can be any type of physical or virtual hardware in the RPS network, for example: virtual machine, computer, virtual switch.

d. Package Stream approval via PowerShell or the Web UI. For more information see How to Approve and Reject Patch Streams.

# How to add a Package to a Package Stream

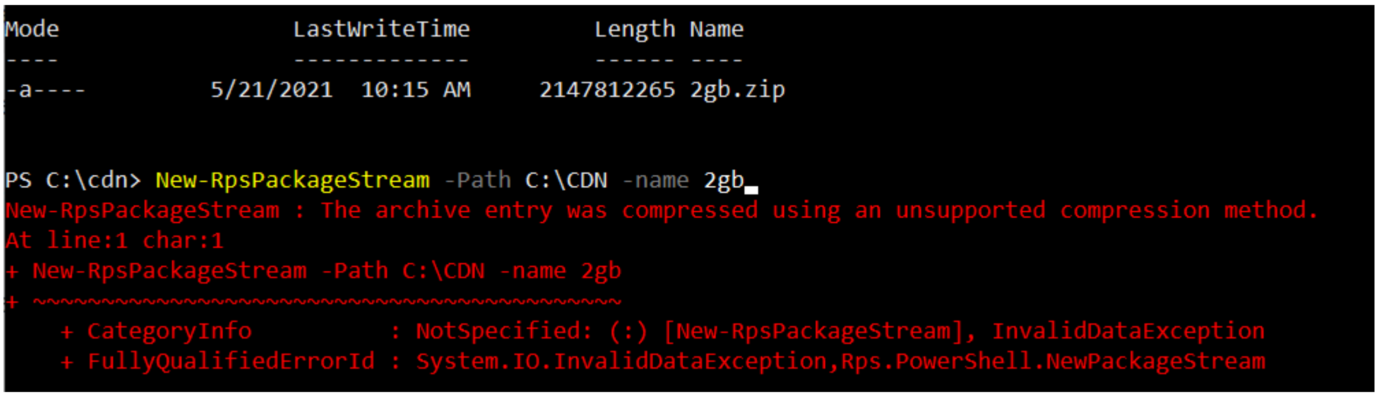
This document describes the step by step instructions for adding a Package to a Package Stream.

## Known Limitations

The maximum supported RPS Package size is 2GB. Any RPS Package zip file that is larger than 2GB will throw an exception when RPS tries to open the Package and read the manifest file from the Package zip file.

- This exception can occur in two scenarios:
  - When creating a new Package Stream with a Package where the zip file size is greater than 2GB.
  - When adding a new Package to an existing Package Stream where the Package zip file size is greater than 2GB.

  **The resulting behavior:**

```
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----          5/21/2021  10:15 AM     2147812265 2gb.zip


PS C:\cdn> New-RpsPackageStream -Path C:\CDN -name 2gb
New-RpsPackageStream : The archive entry was compressed using an unsupported compression method.
At line:1 char:1
+ New-RpsPackageStream -Path C:\CDN -name 2gb
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [New-RpsPackageStream], InvalidDataException
    + FullyQualifiedErrorId : System.IO.InvalidDataException,Rps.PowerShell.NewPackageStream
```

## How to add a Package to a Package Stream using PowerShell cmdlets

1. Open a PowerShell window
2. Change your directory to the RPS ContentStore. For example:

```
cd C:\ContentStore
```

3. Import the RPS API module

```
Import-Module C:\ContentStore\Modules\Rps-Api
```

4. Add the Package using the New-RpsPackage cmdlet.
   - Options for the New-RpsPackage cmdlet are:

| PARAMETER NAME | TYPE | DESCRIPTION |
|---|---|---|
| Path | string | Path of the Package |
| PackageStream | string | Package Stream object that the Package will be added to |

   - Example:

```
New-RpsPackage -Path C:\Packages\mypackage.zip -PackageStream $myPackageStream1
```

5. After the Package command has been executed the following will happen:

a. The Package item will be added to the CMDB

b. The Package file will be copied in to the CDN directory, so it is able to replicate across the CDN.

# How to remove a Package from a Package Stream

This document describes the step by step instructions for removing a Package from a Package Stream.

## How to remove a Package from a Package Stream using PowerShell cmdlets

**NOTE**

This will not remove a Package from the CDN or remove any assignments that may have been made between the Package and any Targets unless the `ForceAssignmentRemoval` switch parameter is used.

1. Open a PowerShell window
2. Change your directory to the RPS ContentStore. For example:

   ```
   cd C:\ContentStore
   ```

3. Import the RPS API module

   ```
   Import-Module C:\ContentStore\Modules\Rps-Api
   ```

4. Remove the Package using the Remove-RpsPackage cmdlet

   - Options for the Remove-RpsPackage cmdlet

     | PARAMETER NAME | TYPE | DESCRIPTION |
     |---|---|---|
     | Name | string | Name of the Package |
     | Package | string | The Package object |
     | ForceAssignmentRemoval | Switch | Indicates whether to remove the Package assignments |

   - Example:

     ```
     Remove-RpsPackage -Name myPackage1 -ForceAssignmentRemoval
     ```

# How to remove a Package Stream

This document describes the step by step instructions for removing a Package Stream.

## How to remove a Package Stream using PowerShell cmdlets

**NOTE**

This will not remove the Packages from the CDN or remove any assignments that may have been made between the Packages and any Targets unless the `ForceAssignmentRemoval` switch parameter is used.

1. Open a PowerShell window
2. Change your directory to the RPS ContentStore. For example:

   ```
   cd C:\ContentStore
   ```

3. Import the RPS API module

   ```
   Import-Module C:\ContentStore\Modules\Rps-Api
   ```

4. Remove the Package Stream using the Remove-RpsPackageStream cmdlet

   - Options for the Remove-RpsPackageStream cmdlet

     | PARAMETER NAME | TYPE | DESCRIPTION |
     |---|---|---|
     | Name | string | Name of the Package |
     | Package | string | The Package object |
     | ForceAssignmentRemoval | Switch | Indicates whether to remove the Package assignments |

   - Example:

     ```
     Remove-RpsPackageStream -Name MyPackageStream1 -ForeceAssignmentRemoval
     ```

# Sideloading a package

When a node has its Cdn communication disabled but needs to get content from its parent it will use Sideloading.

## Process

When a parent node creates a new package it will create all the required CMDB data. If Cdn is disabled then the content wont move to the local node but the CMDB data will. Once the local node has this data it can get the path in the cdn to store the package.

### Get the file hash of package

```
$filePath = 'C:\Packages\Firefox 70.0.zip'

# Calculate the file hash to get the resource id
$resourceId = [Guid]::new([Rps.Api.Utils.HashingUtils]::ComputeMD5Hash($filePath))
```

When a package is created we make the Id the MD5 hash of the package.

### Get the Folder Resource Id

```
# Get the folder Id. FolderId is the reverse of the content resource item
$folderId = [Rps.Api.Utils.HashingUtils]::Reverse($newContent.Id)
```

The folder id is the reverse guid of the Resource Item.

### Join paths together

```
# Get Cdn Folder
$localNode = Get-RpsLocalNode
$cdnPath = $localNode.CdnPath

# get cdn folder path to store file
$cdnFolderPath = Join-Path -Path $cdnPath -ChildPath $folderId
```

Now that we have the full path to the local cdn we will just need to copy the contents over.

### Copy package to CDN

```
# Copy item
Copy-Item -Path $filePath -Destination $cdnFolderPath -Force
```

## Example

```
$filePath = 'C:\Packages\Firefox 70.0.zip'

# Calculate the file hash to get the resource id
$resourceId = [Guid]::new([Rps.Api.Utils.HashingUtils]::ComputeMD5Hash($filePath))

# Get the folder Id. FolderId is the reverse of the content resource item
$folderId = [Rps.Api.Utils.HashingUtils]::Reverse($newContent.Id)

# Get Cdn Folder
$localNode = Get-RpsLocalNode
$cdnPath = $localNode.CdnPath

# get cdn folder path to store file
$cdnFolderPath = Join-Path -Path $cdnPath -ChildPath $folderId

# Copy item
Copy-Item -Path $filePath -Destination $cdnFolderPath -Force
```

# How to Approve and Reject Patch Streams

## Patch Streams

RPS Patch Streams are a collection of one or more patches to be applied to targeted items during a maintenance window.

All steps in this documentation described in the UI will be performed in the RPS Website, on the Packaging Page, under the Approvals tab.

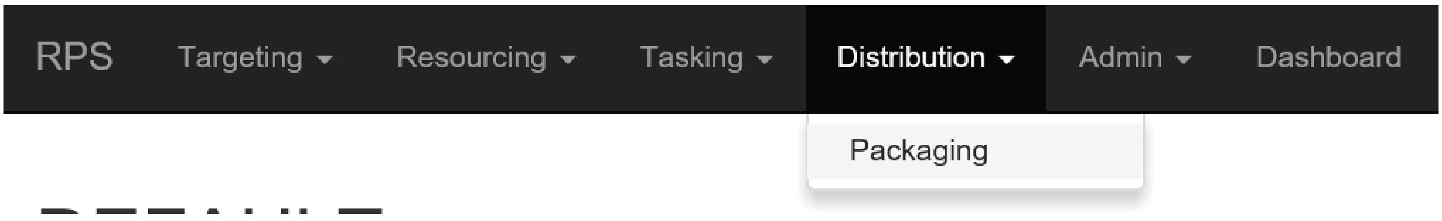


Figure 1: Packaging Page

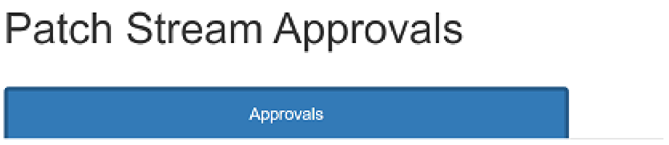


Figure 2: Approvals

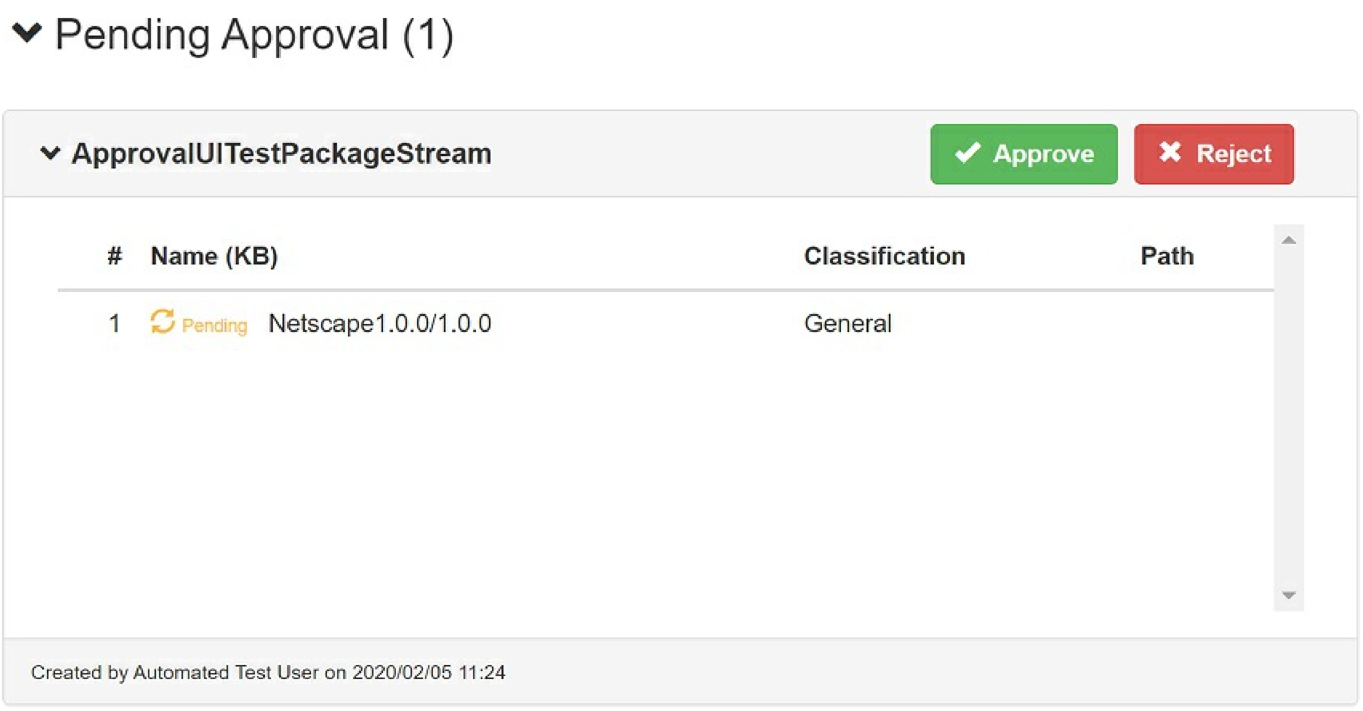
### Approving Patch Streams

The Patch Stream Approver can determine that the Patch Stream can be deployed to applicable target items in some future, Maintenance Window defined, time by Approving the Patch Stream.

From the UI

Using the RPS GUI, Patch Stream Approvers can approve Patch Streams for future deployment.

1. As outlined in the introduction, launch the RPS Website, navigate to the Package Page, and select the "Approvals" tab.
2. When there are no Patch Streams requiring approval, a message stating "*Looks like you have zero Patch Streams waiting for your Approval.*" *will be shown.*
3. When there are Patch Streams requiring approval, they can be approved by clicking the Patch Stream Approval button on the top-right of a Patch Stream panel:



| # | Name (KB) | Classification | Path |
|---|---|---|---|
| 1 | Pending Netscape1.0.0/1.0.0 | General | |

Created by Automated Test User on 2020/02/05 11:24

Figure 3: Pending Approval

1. The Patch Stream will move to the "Scheduled For Deployment" portion of the page.

From PowerShell

In PowerShell, we can Approve a Patch Stream by using the Update-RpsPackageStream cmdlet.

1. Specify the Patch Stream's name in the -Name parameter, and a 'Approved' PackageApprovedStatus to the -ApprovedStatus parameter:

```
$myPackageStream = Update-RpsPackageStream -Name 'MyPackageStream' -ApprovedStatus 'Approved'
```

The cmdlet will return a RPS PackageStream object.

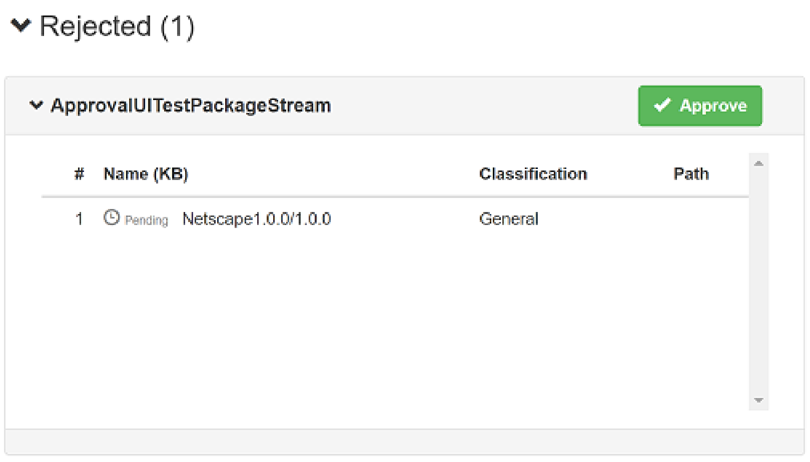
## Rejecting Patch Streams

The Patch Stream Approver can determine that the contents of the Patch Stream are not to be deployed to target items by Rejecting the Patch Stream.

From the UI

Using the RPS GUI, Patch Stream Approvers can approve Patch Streams for future deployment.

1. As outlined in the introduction, launch the RPS Website, navigate to the Package Page, and select the "Approvals" tab.
2. When there are no Patch Streams requiring approval, a message stating "*Looks like you have zero Patch Streams waiting for your Approval.*" will be shown.
3. When there are Patch Streams requiring approval, the Patch Stream can be rejected by clicking the Patch Stream Rejection Button on the top-right of a Patch Stream panel (see Figure 3 Pending Approval)
4. Once rejected, a Patch Stream Approver can optionally choose to Approve the previously rejected Patch Stream by scrolling to the "Rejected" portion of the page and selecting the Patch Stream Approval button:


Rejected (1)

ApprovalUITestPackageStream

| # | Name (KB) | Classification | Path |
|---|---|---|---|
| 1 | Pending Netscape1.0.0/1.0.0 | General | |

Figure 4: Rejected

From PowerShell

In PowerShell, we can Reject a Patch Stream by using the Update-RpsPackageStream cmdlet.

1. Specify the Patch Stream's name in the -Name parameter, and a 'Rejected' PackageApprovedStatus to the -ApprovedStatus parameter:

   ```
   $myPackageStream = Update-RpsPackageStream -Name
   'MyPackageStream' -ApprovedStatus 'Rejected'
   ```

The cmdlet will return a RPS PackageStream object.

## Under The Hood: How patches are assigned to targets

When building your package manifest, the Patch will be matched with destination Target Items on OsVersion, OsType, and Architecture.

For more information on building the package manifest see RPS Packaging 2.1.2 Structure of the Package Manifest File.

# Viewing Patch Stream Deployment Telemetry

## Patch Stream Telemetry Details

There are four statuses that detail the Patch Stream deployment. These statuses are a summation of the status of all of the patches within the stream.

- Pending
- Processing
- Successful
- Error

See Under the Hood: How statuses are evaluated for information on what these terms mean and how those properties determine a Patch's deployment status and renders in the RPS GUI.

## Patch Assignment Telemetry Details

There are numerous statuses that detail the state of a patch and its deployment to an individual assignment:

- Pending
- Processing
- Error
- (Successful) IsPresentAndDesirePresent
- (Successful) IsAbsentAndDesireAbsent
- (Failed) IsPresentAndDesireAbsent
- (Failed) IsAbsentAndDesirePresent

See Under the Hood: How statuses are evaluated for information on what these terms mean and how those properties determine a Patch's deployment status and renders in the RPS GUI.

These statuses can be applied to a Patch Stream via interacting with them in the RPS GUI, or via PowerShell cmdlets described below.

All steps in this documentation described in the UI will be performed in the RPS Website, on the Packaging Page, under the History tab.

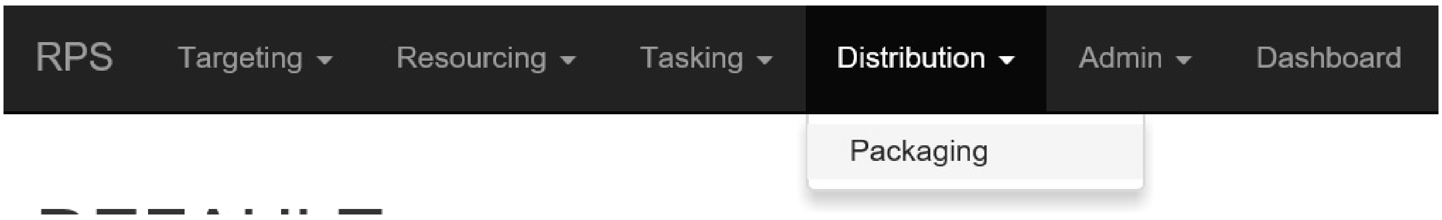


Figure 1: Packaging Page

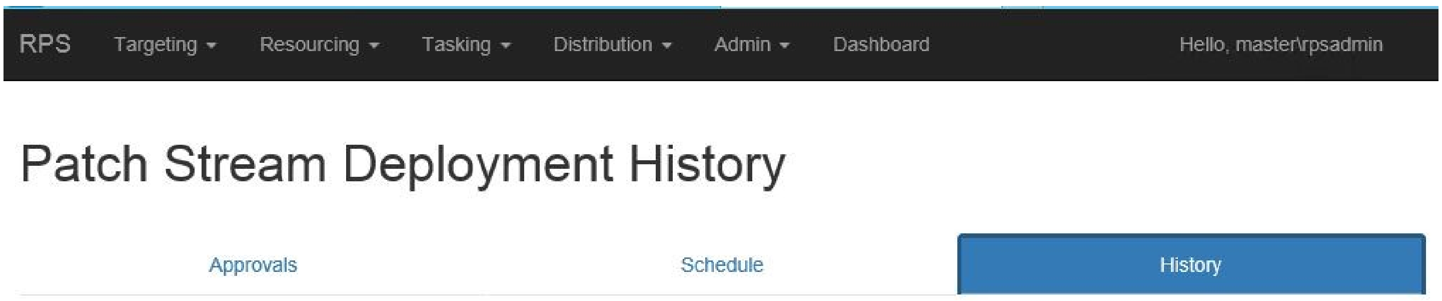


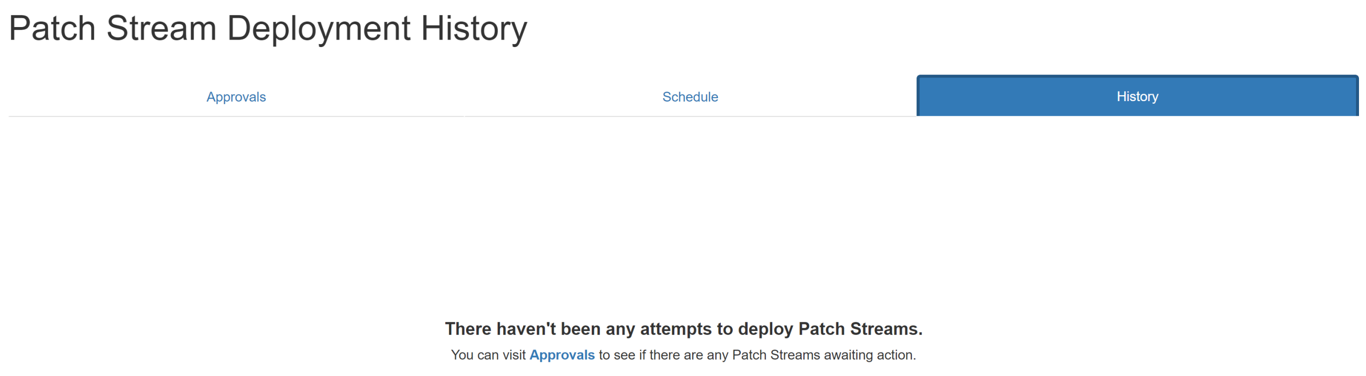
Figure 2: History

## Viewing Patch Stream Telemetry

From the UI

Using the RPS GUI, users can inspect Patch Stream and Patch telemetry from the "History" tab in the Patch Stream Management page of the RPS GUI

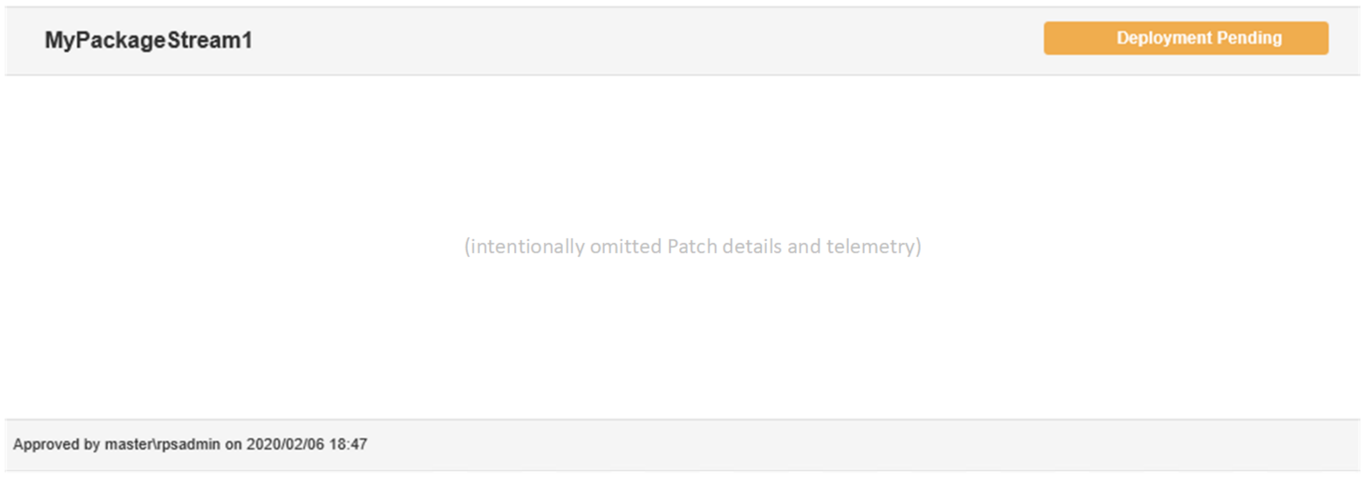
1. As outlined in the introduction, launch the RPS Website, navigate to the Package Page, and select the "History" tab.
2. If there have been no Patch Stream approvals yet, the user should see a message telling them "*There haven't been any attempts to deploy Patch Streams.*"

   **There haven't been any attempts to deploy Patch Streams.**
   You can visit **Approvals** to see if there are any Patch Streams awaiting action.

   Figure 3: Deployment History

3. If there are Patch Streams that have been previously Approved, the user will see a list of them on the screen and the overall deployment status for that Patch Stream will be in the top-right of the panel:

   Approved by master\rpsadmin on 2020/02/06 18:47

   Figure 4: Patch Stream Approval

   In Figure 4, the Patch Stream Deployment Status is "Deployment Pending"

From PowerShell

In PowerShell, users can inspect Patch Stream status using the RPS cmdlet Get-RpsPackageStream.

1. Retrieve the patch stream by specifying Patch Stream's name in the `-Name` parameter of the `Get-RpsPackageStream` cmdlet:

   ```
   $myPackageStream = Get-RpsPackageStream -Name 'MyPackageStream1'
   ```

2. You can print the Approval Status and approval metadata accessing the `ApprovalStatus`, `ApprovedOn`, and `ApprovedBy` properties:

   ```
   Get-RpsPackageStream -Name 'MyPackageStream1' | Select-Object ApprovedStatus, ApprovedOn, ApprovedBy
   ```

## Viewing Patch Telemetry

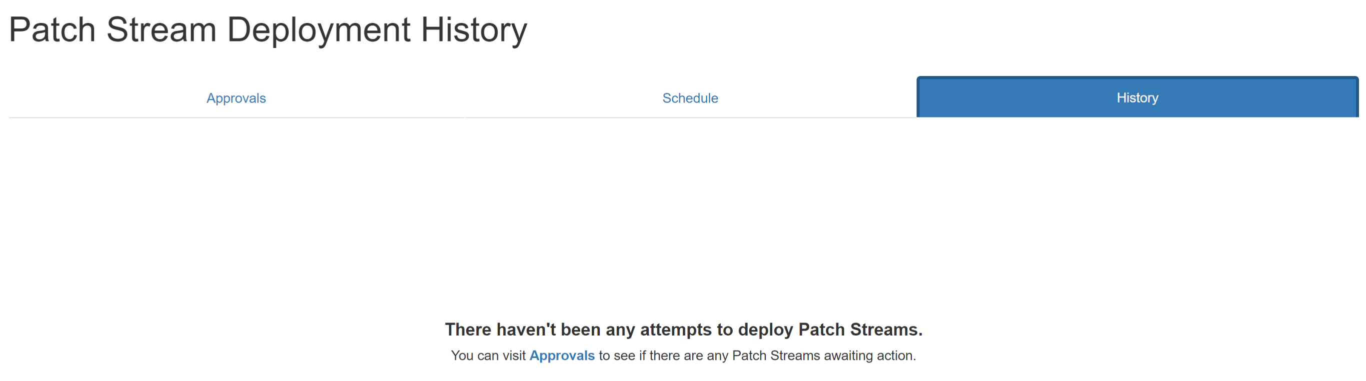
A user serving in the Patch Stream Approver role can determine that the contents of the Patch Stream are not to be deployed to target items by Rejecting the Patch Stream.

From the UI

Using the RPS GUI, Patch Stream Approvers can approve Patch Streams for future deployment.

1. As outlined in the introduction, launch the RPS Website, navigate to the Package Page, and select the "History" tab.

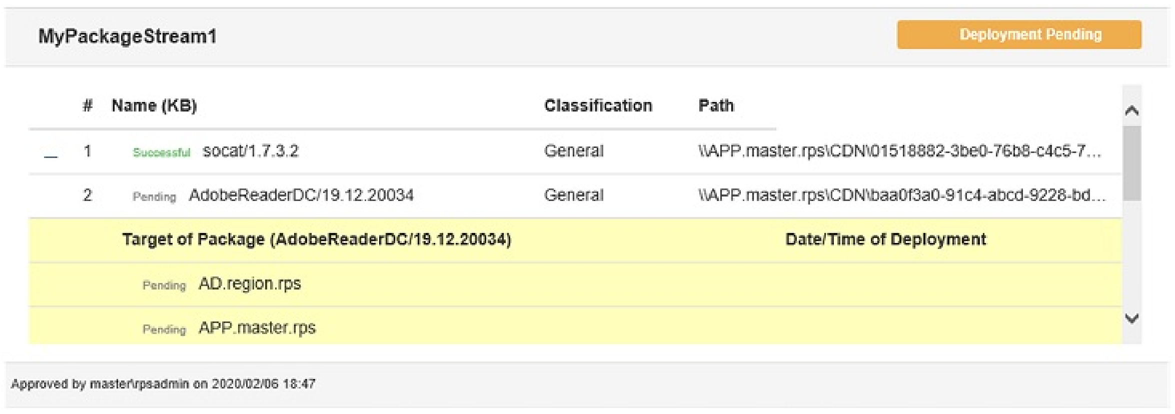
2. If there have been no Patch Stream approvals yet, the user should see a message telling them "*There haven't been any attempts to deploy Patch Streams.*"



There haven't been any attempts to deploy Patch Streams.
You can visit Approvals to see if there are any Patch Streams awaiting action.

Figure 5: Deployment History

3. If there are Patch Streams that have been previously Approved, the user will see a list of them on the screen and the overall deployment status for that Patch Stream, the Patches within the Patch Stream, and the telemetry about the Patches and their deployments to applicable Target Items:



| # | Name (KB) | Classification | Path |
|---|---|---|---|
| 1 | Successful socat/1.7.3.2 | General | \\APP.master.rps\CDN\01518882-3be0-76b8-c4c5-7... |
| 2 | Pending AdobeReaderDC/19.12.20034 | General | \\APP.master.rps\CDN\baa0f3a0-91c4-abcd-9228-bd... |

| Target of Package (AdobeReaderDC/19.12.20034) | Date/Time of Deployment |
|---|---|
| Pending AD.region.rps | |
| Pending APP.master.rps | |

Approved by master\rpsadmin on 2020/02/06 18:47

Figure 6: Patch Stream Approval

### From PowerShell

In PowerShell, users can inspect Patch Stream and Patch telemetry RPS cmdlets Get-RpsPackageStream and Get-RpsPackage.

1. Retrieve the Patch Stream by specifying Patch Stream's name in the -Name parameter of the Get-RpsPackageStream cmdlet:

```
$myPackageStream = Get-RpsPackageStream -Name 'MyPackageStream1'
```

2. Retrieve the Patches by inspecting that Patch Stream's Packages property

```
$myPackageStream.Packages | Select-Object Id, PackageResourceItem
```

3. Retrieve the desired Patch by using the Get-RpsPackage cmdlet, specifying the -Id parameter

```
$myPackage = Get-RpsPackage -Id '<GUID of Patch>'
```

4. Then, retrieve the status of all of the assigned Target Item deployments by inspecting the Patch Assignments' `DeployedStatus`, `Ensure`, and `EndPointState` properties

```
$myPackage.Assignments | Select-Object DeployedStatus, Ensure, EndPointState
```

5. See Under the Hood: How statuses are evaluated for information on what these terms mean and how those properties determine a Patch's deployment status and renders in the RPS GUI.

## Under the Hood: How statuses are evaluated

### How Streams, Patches, and Assignments derive their status

The Patch Stream Status and Patch Deployment Status are roll-up measures of how a Target Item reports up the successful configuration of a Patch.

A Target Item can be assigned a Patch via a RPS Package Assignment (a templatized Resource Assignment).

Within that assignment, RPS tracks the desired state of the Patch (Present (installed) or Absent (uninstalled)).

Whenever the Target Item attempts to reach the desired state, it reports that status all the way back up to the Parent Node CMDB via the Resource Assignment.

The Target Items will report these statuses on the Patch deployment:

- Pending
- Processing
- Error
- (Successful) IsPresentAndDesirePresent
- (Successful) IsAbsentAndDesireAbsent
- (Failed) IsPresentAndDesireAbsent
- (Failed) IsAbsentAndDesirePresent

The Patch within the Stream updates its status as Successful when all the Target Items have successfully reached the desired state (IsPresentAndDesirePresent or IsAbsentAndDesireAbsent) for the assigned Patch.

The Patch Stream updates its status as Successful when all the Patches within it have reached Successful.

## Patch assignment and status flow diagram

See the diagram at "RPS Package Management Workflow:" for more information on how Patches are assigned.

# Enable/Disable CDN Communication

Using PowerShell or the Rps Gui you can enable or disable Bits and Dfsr communication between nodes.

## Dfsr

When DFSR is disabled, all replication groups, memberships, and connections will be created but the connections will be configured as disabled, so no DFSR replication will occur.

## Bits

- When Bits is disabled at the parent node, then then parent will not send its catalog to its children and the child nodes will not get the data to investigate what changes have occurred.
- When Bits is disabled on the local node, then the parent will send its catalog to the local node (if the parent is still enabled). The local node will not process any request coming in related to CDN.
- Any existing jobs that are still in process will resume like normal.

## Enable/Disable CDN from PowerShell

Disable Bits

```
Enable-RpsCdn -Bits $false
```

Disable Dfsr

```
Enable-RpsCdn -Dfsr $false
```

Disable Bits and Dfsr

```
Enable-RpsCdn -Bits $false -Dfsr $false
```

Enable Bits

```
Enable-RpsCdn -Bits $true
```

Enable Dfsr

```
Enable-RpsCdn -Dfsr $true
```

Enable Bits and Dfsr

```
Enable-RpsCdn -Bits $true -Dfsr $true
```

Enable Bits and Disable Dfsr

```
Enable-RpsCdn -Bits $true -Dfsr $false
```

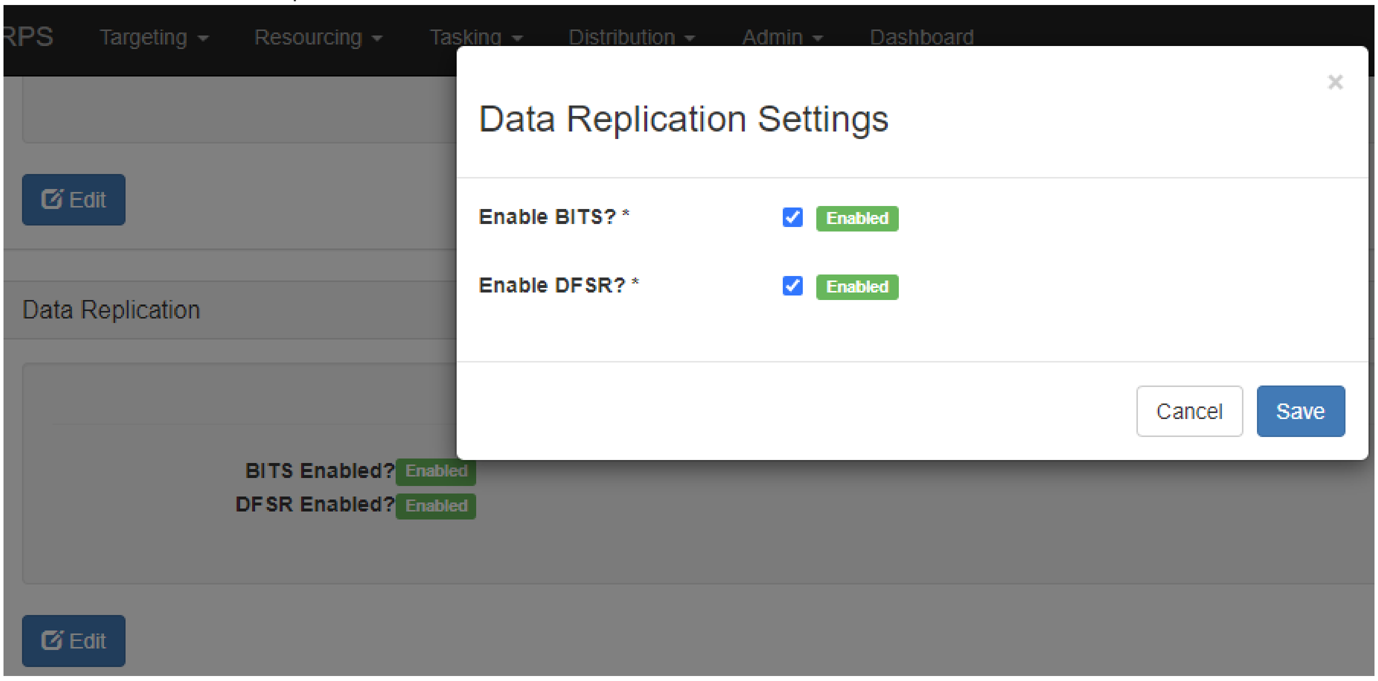
Disable Bits and Enable Dfsr

```
Enable-RpsCdn -Bits $false -Dfsr $true
```

## Enable/Disable CDN from Rps UI

- Access website: https://<server hosting Rps UI>:8080
- Go to Your local node.
- Under Data Replication click 'Edit'

- Check/UnCheck each component of CDN.



Data Replication Settings

Enable BITS? * ✓ Enabled

Enable DFSR? * ✓ Enabled

BITS Enabled? Enabled

DFSR Enabled? Enabled

## Notes

- When configuring the CDN the Bits' and 'Dfsr' properties are internal properties. To disable the Cdn for the entire node you will have to configure the local node. On each local node we will need to add two properties
  - ParentInternal
    - Boolean value that lets Rps know if its parent is internal to the unit.
  - ChildrenInternal
    - Boolean value that lets Rps know if all of its children are internal to the unit.

# Microsoft Server Message Queue (MSMQ)

## Requirement

MSMQ is used as a message queue for communication between RpsSync Service and the BitsDownloaderService.

RpsSync service collects and investigates what files it should download from its parent. Bits jobs need to be created by a user logged in, the RpsSync service cannot create and run the bits jobs. Rps uses MSMQ to store the configuration for the bits job and BitsDownloaderService will pull from the message queue and act on the information in the message.

## Notes

- MSMQ is configured as a Private queue (Active Directory is not used in this instance).
- Permissions given only to the Sync service account and the system account.
- The name of the private queue is called 'BitsQueue'.