Partner Guide for writing extension handlers

Azure VM Agent Team



**The contents of this document are under Microsoft NDA.**

Table of Contents

[1 Overview 2](#_Toc399765471)

[1.1 Terminology 2](#_Toc399765472)

[1.2 Requirements 2](#_Toc399765473)

[1.3 Architecture Overview 3](#_Toc399765474)

[2 Design Details 4](#_Toc399765475)

[2.1.1 Handler Artifacts 4](#_Toc399765476)

[2.2 Handler Lifecycle management 9](#_Toc399765477)

[2.2.1 Add a new handler on the VM (Install and Enable) 9](#_Toc399765478)

[2.2.2 Remove a handler from the VM (Disable and Uninstall) 9](#_Toc399765479)

[2.2.3 Disable 10](#_Toc399765480)

[2.2.4 Enable 10](#_Toc399765481)

[2.2.5 Update 10](#_Toc399765482)

[2.3 Reporting Status and Heartbeat 11](#_Toc399765483)

[2.3.1 Heartbeat reporting 11](#_Toc399765484)

[2.3.2 Status reporting 13](#_Toc399765485)

[2.4 DRAFT (CRP-only): Handler support for multiple extensions 17](#_Toc399765486)

[2.4.1 Handler manifest 17](#_Toc399765487)

[2.4.2 Handler config files 17](#_Toc399765488)

[2.4.3 Handler status files 18](#_Toc399765489)

[3 Aggregate Status Reporting from the VM: 19](#_Toc399765490)

[4 Logging 24](#_Toc399765491)

[5 Registering via Azure Runtime team. 25](#_Toc399765492)

[5.1 Subscription for Registration: 25](#_Toc399765493)

[5.2 Definition file 25](#_Toc399765494)

[6 Self-Publishing using APIs 27](#_Toc399765495)

[6.1 Register API: 27](#_Toc399765496)

[6.1.1 Request Body 27](#_Toc399765497)

[6.2 Update API 29](#_Toc399765498)

[6.3 Updating Extension 29](#_Toc399765499)

[Here is an overview of updates are done: 29](#_Toc399765500)

NOTE: Any questions, feedback, assistance with registration, etc. can all be sent to [azextsup@microsoft.com](mailto:azextsup@microsoft.com)

# Overview

In order to make the Windows Azure IaaS VMs customizable, Windows Azure is releasing a set of capabilities which will enable users to automate software deployment and configuration on IaaS VMs. As a part of these capabilities, a protocol is being released which can be used by various existing VM customization products to integrate with the Windows Azure VM ecosystem. This document discusses the requirements to participate in Windows Azure VM ecosystemand provides a guide for integrating VM customization products with Windows Azure.

## Terminology

|  |  |
| --- | --- |
| Windows Azure Agent | The Windows Azure component that runs inside the VM and is responsible for managing the extension handlers. |
| Handlers | Partner authored component to deliver software and configuration to the customer VM. This component needs to implement handler configuration and status contracts and be provided as a handler package. Generally a handler will consist of an Azure interoperability wrapper around an existing VM customization product.  In the overview documents handlers are more broadly referred to as ‘extensions’. The term ‘handler’ and ‘extension’ are used somewhat interchangeably. |
| Extension Pack | Specific job, workload, or script to be executed by the extension handler. |
| Handler identity | An identifier used to uniquely define the handler. This identity is a tuple of <Handler name>, <Publisher> and <Version> |
| Handler Manifest | A JSON based manifest that defines various properties needed by the Azure Agent to manage the handler. |

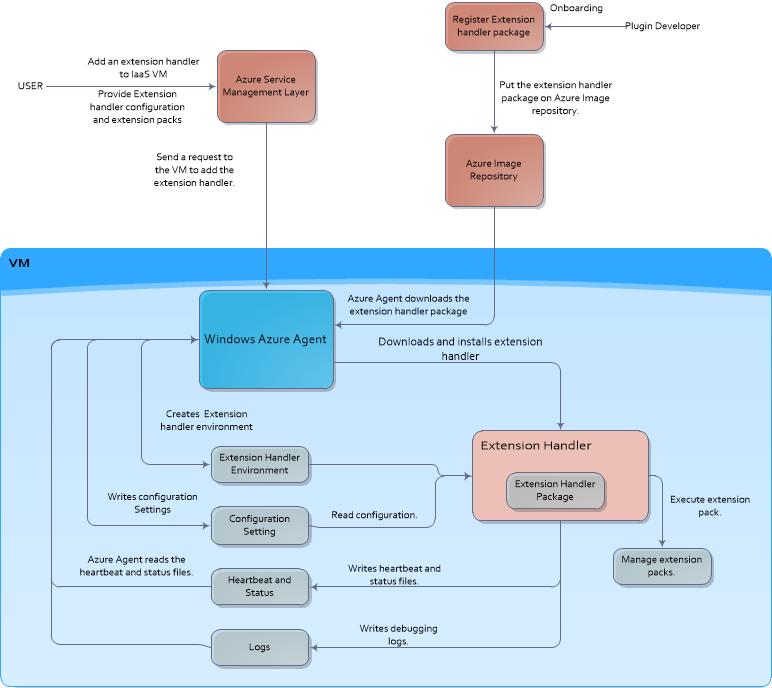
## Requirements

To participate in the Windows Azure ecosystem, any VM customization product needs to create a handler that implements the Windows Azure Agent defined protocol to integrate with the Azure ecosystem. The basic requirements for creating a handler that implements the Azure protocol are:

1. Handler Packaging – The Handler should be packaged as a zip file. This zip package should contain all the binaries related of the handler and HandlerManifest. This package needs to be registered with the Azure image repository. Azure image repository is responsible for managing all versions of all the handlers that are registered with the Azure ecosystem.
2. Handler Environment - Handler needs the capability to read the environment file in the format that Azure Agent defines. The environment file defines the locations of various files and folder that the handler needs to use for reading configuration and writing back heartbeat and status.
3. Handler Configuration – Various extension packs that the handler needs to manage are passed to the handler in form of configuration settings. For example if a script is needed by the handler to install an extension, that script is passed to it via the handler configuration file. The handler should have the ability to read this file in the format defined by the Azure Agent and should be able to execute its contents and report the status of that execution with a frequency that complies with the Azure Agent protocol.
4. Handler heartbeat and status – The handler is supposed to report the status of the most recently executed configuration with a frequency that complies with the Azure Agent protocol. In addition to status, if the handler opts into reporting heartbeat it needs to report the heartbeat for the complete lifetime of the handler on the VM with a frequency that complies with the Azure Agent protocol.

## Architecture Overview

The below diagram gives an overview of how the handlers are supposed to interact with the Azure ecosystem.



# Design Details

### Handler Artifacts

An Azure Extension Handler is composed of the following artifacts:

1. **Handler Package**: This is the package that contains your Handler binary files and all standard static configuration files. This package is registered with the azure ecosystem.
2. **Handler Environment**: This is the set of files and folders that the Azure Agent sets up for the Handlers to use at runtime. These files can be used for communicating with the Azure Agent (heartbeat and status) or for writing debugging information (logging). The details of handler environment created by the Azure Agent is discussed in section 2.1.1.2 below.
3. **Handler Configuration**: This is a configuration file that contains various settings needed to configure this Handler at runtime. Extension configuration is the input provided by the end user based on the schema provided by the handler publisher during registration. For example, a handler might get the client authentication details for writing logs to his storage account via the handler configuration.­

#### Handler Package

The Handlers are packaged as simple zip files for being registered in the Azure ecosystem. The zip file is supposed to contain the following:

* The handler binaries.
* **HandlerManifest.json** file that is used by the Azure Agent to manage the handler. This HandlerManifest.json file should be located in the root folder of the zip file.

The JSON file should be of the format:

[{

"version": 1.0,

"handlerManifest": {

"installCommand": "<your install command>",

"uninstallCommand": "<your uninstall command>",

"updateCommand": "<your update command>",

"enableCommand": "<your enable command>",

"disableCommand": "<your disable command>",

"rebootAfterInstall": <true | false>,

"reportHeartbeat": <true | false>,

"updateMode": <UpdateWithoutInstall | UpdateWithInstall>

}

}]

The above JSON file provides a list of all commands that will be executed by Azure Agent for managing various handlers on the VM.

**version:** indicates the version of the protocol which should be used by the Azure Agent to deserialize this JSON.

**install\uninstall\update\enable\disable** point to the command line that will be executed by Azure Agent in various scenarios. The paths of the command line provided in HandlerManifest.json should be relative to the root directory of the handler. The current working directory of the handler is the path of the root folder of the handler. All these command lines are launched as LOCAL SYSTEM with administrative privileges.

Note: It is valid for multiple commands in the HandlerManifest to point to the same command line. For e.g. the install and Update command might point to the same MSI with same parameters.

**rebootAfterInstall** notifies the Azure Agent if a reboot is required to complete the installation of a handler. Handlers should not reboot the system independently to avoid interfering with each other.

**NOTE : Reboot after install is currently not working. The temporary workaround is:**

In the install command, set a reg key to tell the enable command that it needs a reboot. And then in the enable when you see this, reset the entry and reboot.  
  
You should not reboot from install cmd as it will result in script failure and after reboot install is retried. In enable, you can reboot and return with error code and after reboot the enable will be called.

**reportHeartbeat** indicates the Azure Agent if the handler will be reporting heartbeat or not. The details of heartbeat and status is discussed in section 2.3.1 below.

**updateMode** (optional – default is UpdateWithoutInstall) specifies which update process to follow for the handler. By default, UpdateWithoutInstall is used where during update process install script for the new plugin is not called. The details of how update process works discussed in section 2.2.5 below.

Note that all of the fields in the JSON specified above are required fields and registration of the handler with Azure will fail if one of these fields in not specified. The explanation of the meaning of various fields in the JSON with respect to the Azure Agent is provided in the below sections.

An example of the directory structure of the zip file for a handler is:

SampleExtensionHandler.zip

|-HandlerManifest.json

|-bin

|-MyHandlerAgent.exe

|-MyHandlerService.dll

|-resources

|-En-us

|-installer

|-MyHandlerInstaller.msi

|-updatescript.cmd

A sample HandlerManifest.json for the above sample handler would be:

[{

"version": 1.0,

"handlerManifest": {

"installCommand": "installer\MyHandlerInstaller.msi",

"uninstallCommand": "installer\MyHandlerInstaller.msi",

"updateCommand": "installer\updatescript.cmd",

"enableCommand": "net start MyHandlerAgent",

"disableCommand": "net stop MyHandlerAgent",

"rebootAfterInstall": false,

"reportHeartbeat": true,

"updateMode": "UpdateWithoutInstall"

}

}]

#### HandlerEnvironment

When the Azure Agent installs a handler on the VM it creates a bunch of files and folders that are needed by the handler at runtime for various purposes. The location of all these files and folders are communicated to the handler via the HandlerEnvironment .json file.

HandlerEnvironment.json is the file that is created under the root directory where the handler is unpackaged. The structure of HandlerEnvironment.json is:

[{

"version": 1.0,

"handlerEnvironment": {

"logFolder": "<your log folder location>",

"configFolder": "<your config folder location>",

"statusFolder": "<your status folder location>",

"heartbeatFile": "<your heartbeat file location>",

"deploymentid": "<deployment id for the vm>",

"rolename": "<role name for the vm>",

"instance": "<instance name for the vm>"

}

}]

version: contains the version of the protocol that the Azure Agent is abiding with. In the initial release the only supported version is 1.0.

handlerEnvironment – This is the object that encapsulates all the properties of a handler defined in the version 1.0 of the protocol.

logFolder – contains the location where the handler should put its log files that might be needed to debug any customer issues. The advantage of putting log files under the folder directed by this location is that these files can be automatically retrieved from the customers VM by using a tool, without actually logging into the VM and copying them over manually.

configFolder – contains the location where the handler will get its configuration settings file.

statusFolder – contains the location where the handler is supposed to write back a file with a structured status of the current state of the work being done by the handler.

heartbeatFile: this is the file that is used to communicate the heartbeat of the handler back to the Azure Agent.

deploymentid: this is the deployment id which the vm belong to

rolename: this is the role name of the vm

instance: this is the instance name of the vm (can be same as role name)

**Errors while reading HandlerEnvironment.json –** In rare cases a handler might encounter errors when trying to read the HandlerEnvironment.json file, since the Azure Agent might be writing the file at the same time as well. The handler should be capable of handling such errors. Our recommendation for handler publishers would be to have a retry logic with some sort of backoff. The handler publisher can also rely on the Helper Library discussed in section 1.6 to avoid handling the file system conflict issues.

#### Handler Configuration

There are scenarios when a handler needs some user input parameters to configure its handler. All such user provided input is communicated from the Azure Agent to the handler via the configuration file. For e.g. a handler might require the user to provide the account name and the key of a user storage account where the logs will be saved. This account information can be passed by the user to the handler via the configuration file.

##### Configuration File Structure

The configuration file should be a valid JSON with the only property of the root object as “runmtimeSettings”. Under runtimeSettings is an array of “handlerSettings” objects and with two child objects “protectedSettings” and “publicSettings”. Apart from that the complete schema of the handler configuration file under the “publicSettings”\”protectedSettings” property is defined by the handler publisher during the registration process. When a call to add the handler to the VM is made, the user needs to provide a configuration that complies with the structure that the handler publisher had provided during registration.

**Managing user secrets:** There may be parts of the handler configuration that contain user secrets (like passwords, storage keys, etc). These secrets in general should never be persisted in plain text to prevent accidently disclosure. To support this concept, the Azure Extension Handler publishers can allow users to store all or part of the handler configuration in a protected section of the config. All settings under this section are encrypted by an X509 certificate before being sent over to the VM. The Azure Agent will persist the protected settings as encrypted only and will provide the thumbprint of the certificate that needs to be used for decrypting this information. To extract the setting, the handler will need to retrieve the certificate from the Local Machine store and decrypt the settings using the certificate private key. The publisher of the Azure Extension Handler decides what, if any, part of the configuration should be protected in this manner.

A sample configuration file would look like:

{

"runtimeSettings": [

{

"handlerSettings": {

"protectedSettings": {

"storageaccountname": "MY SECRET STORAGE ACCOUNT NAME",

"storageaccountkey": "MY SECRET STORAGE ACCOUNT KEY"

},

"publicSettings": {

"MyHandlerConfiguration": {

"configurationChangePollInterval": " ",

"overallQuotaInMB": 12

},

"MyHandlerInfrastructureLogs": {

"scheduledTransferLogLevelFilter": "Verbose",

"bufferQuotaInMB": "100",

"scheduledTransferPeriod": "PT1M"

}

}

}

}

]

}

In the above example the storageaccountname and storageaccountkey are protected secrets. When these secrets are persisted on a file in the VM for consumption by the handler the protected section would be encrypted and base64 encoded. In the case of above settings, the configuration file for the above sample on the VM would look like:

{

"runtimeSettings": [

{

"handlerSettings": {

“protectedSettingsCertThumbprint”: “a811c3f4058542418abb”,

"protectedSettings": “ICB7DQogICAgInN0b3JhZ2VhY2NvdW50IiA6ICJbcGFyY

W1ldGVycy5TdG9yYWdlQWNjb3VudF0iLA0KICB9LA0K”,

"publicSettings": {

"MyHandlerConfiguration": {

"configurationChangePollInterval": " ",

"overallQuotaInMB": 12

},

"MyHandlerInfrastructureLogs": {

"scheduledTransferLogLevelFilter": "Verbose",

"bufferQuotaInMB": "100",

"scheduledTransferPeriod": "PT1M"

}

}

}

}

]

}

The code in C# that can be used to decrypt the handler is:

public static string DecryptProtectedSettings(string encryptedSettings, X509Certificate2 cert)

{

var envelope = new EnvelopedCms();

envelope.Decode(Encoding.UTF8.GetBytes(encryptedSettings);

X509Certificate2Collection certs = new X509Certificate2Collection(cert);

envelope.Decrypt(certs);

var decryptedBytes = envelope.ContentInfo.Content;

return Encoding.UTF8.GetString(decryptedBytes); // this is the value of the protectedSettings property as a JSON object.

}

##### Location of Handler configuration

The location where the configuration setting files will be written can be retrieved by the “configFolder” property in the **HandlerEnvironment.json** file.

##### Handler configuration filename

Whenever a new configuration is received, Azure Agent will write the configuration settings file named <SequenceNumber>.settings under the configFolder with the configuration provided by the user and launches the enable command of the handler (section 2.2.4).

The handler is expected to retrieve the last sequence number of the configuration file written by Azure Agent bylooking under the configfolder directory for the highest sequence number.. This sequence number can then be used to apply the latest user provided configuration settings to the handler.

## Handler Lifecycle management

### Add a new handler on the VM (Install and Enable)

When a handler is requested on a VM by the user, the Azure Agent will do the following inside the VM:

* Download the handler package zip from the azure repository.
* Unzip the package under a unique location corresponding to the handler identity. The handler should **not** take any dependency on the location where the handler package is unpacked, since this location might change in future depending on future requirements.
* Create the configuration, logging and status folders for the handler.
* Creates the HandlerEnvironment.json file under the root folder where the handler is unpacked.
* Parse the HandlerManifest.json file and execute the install command in a separate process.
* The install command is executed in the process with the administrative privileges.
* If there are multiple handlers that are being installed the Azure Agent will download and unzip them in parallel but will invoke the install command sequentially only.
* The Azure Agent will wait for the installation to complete and monitor the exit code of the install process.
* If the install process exits SUCCESSFULLY (exit code 0), Azure Agent maintains state that the handler was installed successfully and does not run the install command for the same handler again ever unless the handler has been uninstalled first.
  + Azure Agent will wait for a maximum of 15 minutes before timing out the install process and considering the install to be failed.
* If the install process exits SUCCESSFULLY, Azure Agent will provide the handler configuration settings in the defined location and launch the Enable command in a separate process that runs as LOCALSYSTEM with ADMINISTRATIVE privileges.
  + Azure Agent will wait for a maximum of 5 minutes before timing out the enable process and considering the enable to be failed
* If the install process exits UNSUCCESSFULLY the Azure Agent will retry to install the handler under two circumstances:
  + When the Azure Agent receives a new goal state triggered by a user action. (e.g. Adding\removing\updating any handler or updating handler configuration etc.)
  + When the Azure Agent restarts (which should only happen when the machine itself is rebooted).

**Note: Runtime config settings file will be placed after install is finished and will be available for the enable script**

### Install command

In the install command the handler is expected to install its processes and services on the system and create the necessary setup that is required for the handler to run at runtime.

### Remove a handler from the VM (Disable and Uninstall)

When a user explicitly requests to remove the handler from the VM, the Azure Agent will execute the following actions:

* The disable command specified in the HandlerManifest.json will be executed in a separate process that runs as LOCALSYSTEM with ADMINISTRATIVE privileges. The handler is expected to complete the pending tasks and then stop any processes or services related to the handler that have been running on the machine.
  + Azure Agent will wait for a max of 15 minutes for the disable process to finish before timing out to the next steps.
* The uninstall command will be invoked in a separate process that runs as LOCALSYSTEM with ADMINISTRATIVE privileges. Azure Agent will wait for a maximum of 5 mins for the uninstall process to finish.
* Azure Agent will remove all the package binaries and configuration files that were associated with the handler. The handler log files will be maintained on the machine for any future debugging purposes.

### Disable

A user might explicitly request to disable a handler without uninstalling it. On disable Azure Agent will execute the disable command in a separate process with ADMINISTRATIVE privileges. On the execution of the disable command the handler is expected to complete the pending tasks and then stop any processes or services related to the handler that have been running on the machine.

Azure Agent will wait a max of 15 mins for the disable process to finish before timing out to the next steps.

### Enable

A user might explicitly request to enable a handler that has been previously disabled. On enable Azure Agent will execute the enable command in a separate process with ADMINISTRATIVE privileges.

The enable command will be invoked every time the machine reboots or the machine receives a new configuration settings file. Enable will have 5 minutes to complete its task, after which GA will kill the process. If ‘enable’ has to perform any long running tasks, it should be handled in a separate process and the status should be reported to the status channel.

In the 5 minute window the enable script should performs validation and fail if the configuration is in correct or if the necessary components are not installed. If the validation passes, it should launch a separate process for the actual enable process. Any long installations should be handle during install which has a 15 minute window.

**Note:**

Unlike the install state, the Azure Agent will not maintain the enabled\disabled state of the handler. Every time the machine restarts (which in turn will restart the Azure Agent) or a new goal state is received, the Azure Agent will try to set the machine to the latest goal state. Thus it might invoke the enabled\disabled commands multiple times even if the handler is already enabled\disabled. So the enable and disable commands need to be idempotent i.e. if the handler is already enabled and the enable command is invoked again, the command should check if all the processes are running as expected, if yes, then the command should just exit with a success code.

### Update

There are two scenarios when an update can happen:

1. The user triggers an explicit update of the handler.
2. The handler is updated on Azure repository and it automatically gets picked up by the Azure Agent.

In both these cases the Azure Agent will identify that a handler with the same name and publisher and a lower version is already installed on the machine.

1. It will download the updated version of the handler from azure repository, unpack it under the handler identity folder.
2. Azure Agent will call disable on the existing handler with the lower version.
3. Azure Agent will invoke the update command in the newly downloaded packages under a separate process with ADMINISTRATIVE privileges. During update the handler has an opportunity to transfer any state information from the previous handler. Vm agent will wait 15 mins for update script to finish
4. Azure Agent will invoke the uninstall command on the existing handler with lower version.
5. (Conditional step) If handler manifest has updateMode (see 2.1.1.1) specified with UpdateWithInstall then Azure Agent will invoke the install command on thenewly downloaded package. If its not specified or has value UpdateWithoutInstall then this step is skipped
6. Azure Agent will invoke the enable command on the newly downloaded package

## Reporting Status and Heartbeat

Windows Azure provides two facilities to report back the health of the handler and the status of the operations being performed by it.

1. Heartbeat: Heartbeat channel should be used to report the health of the handler itself. Providing heartbeat is an optional facility that the handler can opt into by setting the reportHearbeat property to true in the HandlerManifest. Heartbeat is generally expected to be reported by long running services or processes. For eg, an antivirus handler service might use the heartbeat channel to indicate if its service has stopped for some reason.
2. Configuration Status: Status channel should be used to report the success or failures of any operations that were conducted when applying the new configuration provided by the user. For e.g. Diagnostics agent might report issues connecting to the storage account via this channel.

The Azure Agent collects the heartbeat and status information for all handlers and aggregates them into VM health which is returned to the user when he queries for it via the GetDeployment RDFE API call.

### Heartbeat reporting

The handler that have opted into reporting heartbeat are supposed to report it via the file specified in the heartbeat property of the HandlerEnvironment file. The structure of the heartbeat file should be:

[{

"version": 1.0,

"heartbeat" : {

"status": "<ready | notready>",

"code": <Valid integer status code>,

"message": {

"id": "id of the localized resource",

"params": [

"MyParam0",

"MyParam1"

]

},

"formattedMessage": {

"lang": "Lang[-locale]",

"message": "formatted user message"

}

}

}]

Various fields in the above JSON document correspond to the following:

**version** – This is the version of the protocol being used to communicate heartbeat to the Azure Agent. Currently the only version Azure Agent understands is 1.0.

**heartbeat** – This object encapsulates all the heartbeat related information for the handler.

**status** – The current status of the handler. The only valid values are “ready” and “notready”.

**code** – The status code the handler. This is an optional field.

**heartbeat\message** – This is an optional localized message that will be passed back to the user on a GetDeployment call via RDFE.

**heartbeat\message\id** – This is the message identifier, to be used for lookup of a localized message. Treated as a string. A symbolic id is preferred for human interpretation, for example Error\_CannotConnect. The file that contains all the localized strings corresponding to the id would be provided by the handler author to Azure during registration.

**heartbeat\message\params:** This is an Ordered list of parameter (placeholder) values to be filled into the message template corresponding to the message id. The first Param is used for placeholder “{0}” in the message template (from the provided language resources); the second for placeholder “{1}”, etc.

**heartbeat\formattedMessage\lang:** The language/locale of the preformatted message. Like: en-US

**heartbeat\formattedMessage\message:** The human readable message that will be returned to the user.

Handlers can report successful heartbeat by setting the status to "ready". To report repeated successful heartbeats, the handler can just change the last modified timestamp of this file. The status field only needs to be changed to “notready” if the handler has encountered some error\exception condition while executing. For e.g. If after the handler is installed and before the first configuration settings file is processed, if there is an exception, it can be reported via the status section in the heartbeat file.

Azure Agent will read the heartbeat file once every 2 minutes to check if the plugin is running or not. If the last modified timestamp is within the last 1 minute and the status is set to "ready" then Azure Agent will consider the plugin to be working properly. If the last modified timestamp is older than 10 minutes, Azure Agent will consider the plugin handler to be unresponsive. If the last modified timestamp is between 1 minute and 10 minute, Azure Agent will consider the plugin to be in "Unknown" state. If the status is set to NotReady, the error code and the message will be returned back to the user in the next GetDeployment call.

A sample heartbeat file would look like:

[{

"version": 1.0,

"heartbeat" : {

"status": "ready",

"code": 0,

"formattedMessage": {

"lang": "en-US",

"message": " Sample Handler running. Waiting for a new configuration from user."

}

}

}]

**Errors while writing to the HeartBeat file –** In rare cases a handler might encounter errors when trying to write the heartbeat file, since the Azure Agent might be reading the file at the same time as well. The handler should be capable of handling such errors. Our recommendation for handler publishers would be to have a retry logic with some sort of exponential backoff. The handler can also use the helper library discussed in section 1.6 to avoid handling the filesystem conficts.

### Status reporting

The handler **\*MUST\*** report status (for a passed runtime config) back to Azure Agent by writing to the status file “<SequenceNumber>.status” under the status folder specified in the HandlerEnvironment. The status file structure supported by the azure agent is:

[{

"version": 1.0,

"timestampUTC": "<current utc time>",

"status" : {

"name": "<Handler workload name>",

"operation": "<name of the operation being performed>",

"configurationAppliedTime": "<UTC time indicating when the configuration was last successfully applied>",

"status": "<transitioning | error | success | warning>",

"code": <Valid integer status code>,

"message": {

"id": "id of the localized resource",

"params": [

"MyParam0",

"MyParam1"

]

},

"formattedMessage": {

"lang": "Lang[-locale]",

"message": "formatted user message"

},

"substatus": [{

"name": "<Handler workload subcomponent name>",

"status": "<transitioning | error | success | warning>",

"code": <Valid integer status code>,

"message": {

"id": "id of the localized resource",

"params": [

"MyParam0",

"MyParam1"

]

},

"formattedMessage": {

"lang": "lang[-locale]",

"message": "formatted user message"

},

}]

}

}]

**version** – indicates the version of the protocol being used for communicating the status back to the Azure Agent.

**timestampUTC** – The current time in UTC during which this status structure is being created. This should be in parsable UTC format. E.g: 2013-11-17T16:05:14Z.

**status** – The object that encapsulates the top level status about the configuration corresponding to what the status is being reported.

**status\name** – This property is optional. This property can be used by handlers to point to the VM workload name that are being managed by the handler.

**status\operation** – This property is optional. This property can be used by handlers to indicate the current operation being performed to enable the VM workload on the machine.

**status\configurationappliedtime** – This property is optional. This property can be used by handlers to indicate the last time the configuration corresponding to the current sequence number was successfully applied on the VM. Time needs to be in parsable UTC time. E.g: 2013-11-17T16:05:14Z

**status\status** – This property indicates the current status of the operation being performed. The only acceptable values are: Transitioning, error, success and warning.

**status\code** – A valid integer status code for the current operation.

**status\message** – This is an optional localized message that will be passed back to the user on a GetDeployment call via RDFE.

**status\message\id** – This is the message identifier, to be used for lookup of a localized message. Treated as a string. A symbolic id is preferred for human interpretation, for example Error\_CannotConnect. The file that contains all the localized strings corresponding to the id would be provided by the handler author to Azure during registration.

**status\message\params:** This is an Ordered list of parameter (placeholder) values to be filled into the message template corresponding to the message id. The first Param is used for placeholder “{0}” in the message template (from the provided language resources); the second for placeholder “{1}”, etc.

**status\formattedMessage\lang:** The language/locale of the preformatted message. Like: en-US

**status\formattedMessage\message:** The human readable message that will be returned to the user.

**substatus** – An array of nested substatus objects that can be used by the handler to pass the substatus of complicated operations. The fields in the substatus array are supposed to be used in the same manner as they are used in the parent status array.

Everytime a handler receives a new handler pack via a new configuration, it is expected to periodically report the status corresponding to that configuration in a file names <SequenceNumber.status>. The status should be reported atleast once every 2 minutes for the time when the handler is in (transitioning\Warning) state. Once the handler reaches a terminal state (success\error) it can stop reporting the status messages for that sequence number.

Each time the handler has new status to report, it should overwrite <SequenceNumber.status> file. The status provided in the status file should be an aggregate status (even if that status has been reported before) of all the operation performed for this configuration so far. If writing to the file fails, the handler should retry with backoff. The handler can write to the status file whenever it has something new to report. Azure Agent will only read this status file after it has fed a new configuration to the handler and till the time the handler does not report status of a terminal state (success\error). During this time the Azure Agent will read the status file with a default frequency of 5 mins (configurable).

Note: Here is the recommended practice for troubleshooting any handler errors:

* Always Report status ‘0’ unless there is a bug in the handler itself which prevents it from running. So errors caused by external factors like incorrect configuration or lack of network access should be reported as ‘0’ with details in the log.
* If the handler encounters a critical issue, then report a unique error code for each failure path. These error codes will be collected in the Azure BI channel which can be used for troubleshooting.

A simple status report without localization from a handler would look like:

[{

"version": 1.0,

"timestampUTC": "2013-11-17T16:05:14Z ",

"status" : {

"name": "enable wordpress",

"operation": "installing wordpress",

"status": "transitioning",

"formattedMessage": {

"Lang": "en",

"Message": "Enable IIS on the VM."

},

"substatus": [{

"name": "Wordpress plugin",

"status": "success",

"code": 0,

"formattedMessage": {

"lang": "en-US",

"message": "Successfully downloaded wordpress plugin."

}

},

{

"name": "Enable IIS",

"status": "transitioning",

"message": "Turning windows feature for enabling IIS on."

}

]

}

}]

#### Localization Support

NOTE: Currently the platform **does not** support localized messages. The current recommendation is to use the format specified in the above section for status reporting.

The rest of this section can be skipped for now.

To enable showing these messages in the user’s preferred language and, ideally, to enable multiple users to view the same captured execution status in different languages, we need to defer message resource lookup until the user queries for handler status. The current user’s preferred language would be retrieved from the HTTP header.

Localization support is optional. If the handler does not wish to participate in localization they can just return the FormattedMessage strings in a default language which will be directly returned to the user.

A localized status report would look like:

[{

"version": 1.0,

"timestampUTC": "<current utc time>",

"status" : {

"name": "SharePointFrontEnd",

"operation": "ResExtProvisioning",

"status": "error",

"code": 12,

"message": {

"id": "1215",

"params": [

"spo-sqldb.cloudapp.net", "JoeAdmin"

]

}

}

}]

#### Localized message formatting

As part of handler registration with Azure, a set of localization resources will be provided for looking up the status messages from the handler.

A language/locale lookup sequence similar to the one for [.NET resources](http://msdn.microsoft.com/en-us/library/vstudio/sb6a8618(v=vs.100).aspx) will be applied, with the ultimate fallback being “en”, a resource file for which must always be provided.

The structure of the JSON resource files will be as follows.

[{

"version": 1.0,

"lang":"lang[-locale]",

"messages": [

{

"id": "message id",

"text": "Message text with {0}, {1} placeholder."

}]

}]

**Placeholder ordering**: The order of Status/Param values from the in-guest handler must be fixed (independent of language) and should correspond to the sequence of {n} placeholders in the English version of the message. If translation of a message in some language requires different order of the placeholders, the message template in the resource file for that language should have the placeholders reordered accordingly. To continue the earlier Status sample the message corresponding to id 1215, if in English we have:

Failed to establish connection to {0} as {1}

In German it might be:

{1} fehler beim Anschluss an {0} herzustellen

#### Check Aggregate Status file

After writing the status file, check the latest aggregatestatus.json file at: C:\WindowsAzure\Logs\AggregateStatus. If there is any parsing error for status then check the log file c:\windowsazure\logs\WaAppAgent.log to see full error message.

This the file that will be sent to RDFE, so make sure that the status reported is de-serialized correctly into this file

## Handler support for multiple extensions

With the CRP Preview, it is now possible for handlers to support running multiple extensions at the same point in time. This allows general purpose handlers (such as DSC, Custom Script, Chef, Puppet, VMM) to let users specify multiple extensions to run when provisioning or updating their VMs (such as having one DSC script configure SharePoint and another DSC script install antivirus). This removes a major blocker for writing modular extensions in RDFE (where using one extension for a handler precludes a user from also using another).

This is an optional capability that requires handler opt-in and makes some changes to placement of the configuration and status files. It is up to the handler to decide whether to actually execute the extensions in parallel or to sequence them. Opting into the capability only requires the handler to understand how to find the new settings files and how to maintain the new set of status files.

### Handler manifest

A handler is opted-in to this capability by providing a value of “true” for the “supportMultipleExtensions” property in the handler manifest. If the property is omitted, the default value is “false”.

[{

"version": 1.0,

"handlerManifest": {

"installCommand": "installer\MyHandlerInstaller.msi /q /f",

"uninstallCommand": "installer\MyHandlerInstaller.msi /u",

"updateCommand": "installer\updatescript.cmd",

"enableCommand": "net start MyHandlerAgent",

"disableCommand": "net stop MyHandlerAgent",

"rebootAfterInstall": false,

"reportHeartbeat": true,

"updateMode": "UpdateWithoutInstall",

"supportMultipleExtensions": true,

}

}]

### Handler config files

If opted in to this capability, the handler will no longer see a config file named <SequenceNumber>.settings in the config folder, even when there is only one extension.

Instead, there will be an array of 0 to many settings files following the naming convention <extensionname>.<sequencenumber>.settings. Each file will individually follow the same content schema as outlined in section 2.1.1.3.1 (the file structure does not change when opting in, just the number and names of the files).

The handler is responsible for applying the settings from each of these files but it is left to the handler to determine whether to run serially or in parallel and in what order to apply.

#### Extension Update behavior

When the configuration changes for an individual extension, only the settings file for that extension will be updated. This means that the handler may have different sequence numbers for each of its extensions. The handler is expected to apply the most recent settings file for each extension, even if one extension has a lower sequence number than another.

If a user removes an extension from a VM but there is still one or more other extensions remaining, the handler will simply see the settings file for an extension disappear. This implies that the settings has been removed and status no longer needs to be reported for it.

Here is a sample of the files that the handler should expect to see and operate on after given actions:

|  |  |
| --- | --- |
| **Action** | **Settings File list** |
| User deploys VM with extensions A and B | A.0.settings, B.0.settings |
| User updates extension A on VM | A.1.settings, B.0.settings |
| User updates both extensions A and B | A.2.settings, B.1.settings |
| User removes extension A from VM | B.1.settings |

### Handler status files

Similar to the settings files, a handler is expected to output status files with a new naming scheme when opted into this format, even if there is only a single extension.

The handler should output 1 to many status files following the naming convention <extensionname>.<sequencenumber>.status (1 per settings file received). Each file will individually follow the same content schema as outlined in section 2.3.2 (the file structure does not change when opting in, just the number and names of the files).

#### Update behavior

The handler is essentially responsible for outputting one status file per settings file with the same sequence number as the settings file being executed. The handler should generally delete a status file when the corresponding settings file is removed but the agent does not depend on this behavior (since it will not try to read a status file if there is not a corresponding settings file).

Here is a sample of the files that the handler should expect to see and operate on after given actions:

|  |  |
| --- | --- |
| **Action** | **Status File List** |
| User deploys VM with extensions A and B | A.0.status, B.0.status |
| User updates extension A on VM | A.1.status, B.0.status |
| User updates both extensions A and B | A.2.status, B.1.status |
| User removes extension A from VM | B.1. status |

# Aggregate Status Reporting from the VM:

VMAgent aggregates its own status (every 15-30 secs), along with the status of all the extensions and uploads it to the user storage which can be retrieved by RDFE APIs (in corresponding XML format). VM Agent will overwrite aggregatestatus.json file at: C:\WindowsAzure\Logs\AggregateStatus

{

"version":"1.0",

"timestampUTC": "<Utc time of generated status file>",

"aggregateStatus":

{

"guestAgentStatus":

{

"version":"<version of guest agent running>",

"status":"<NotReady | Ready | Unresponsive>",

"code": <Valid integer status code>,

"message": {

"id": "id of the localized resource",

"params": [

"MyParam0",

"MyParam1"

]

},

"formattedMessage": {

"lang": "Lang[-locale]",

"message": "formatted user message"

},

},

"handlerAggregateStatus":

[

{

"handlerName":"<Name of the reported handler>",

"handlerVersion":"<Version of the reported handler>",

"status":"<Ready | NotReady | Installing | Unresponsive>",

"code": <Valid integer status code>,

"message": {

"id": "id of the localized resource",

"params": [

"MyParam0",

"MyParam1"

]

},

"formattedMessage": {

"lang": "Lang[-locale]",

"message": "formatted user message"

},

"runtimeSettingsStatus": {

"sequenceNumber": "<Sequence number of the runtime config processed>",

"settingsStatus": {

**<Handler status data – see section 2.3.2 excluding “version: 1.0”>**

},

"extensionName": "<Name of the reported extension>"

}

}

,… (more handler’s statuses)

]

}

}

**version** – indicates the version of the protocol being used for communicating the aggregate status back to RDFE from Azure Agent.

**timestampUTC** – The current time in UTC during which this aggregate status structure is being created. This will be in parsable UTC format. E.g: 2013-11-17T16:05:14Z

**aggregateStatus** - The object that encapsulates the top level status about VM Agent and all the handlers on the VM.

**aggregateStatus\guestAgentStatus** – Represents the health of the VM Agent itself.

**aggregateStatus\guestAgentStatus\version** – Version of the guest agent running on the VM

**aggregateStatus\guestAgentStatus\status** – Status of VM Agent.

**NotReady** – Only returned by RDFE when there is no status file uploaded yet. Means VM might not be up yet and hence caller should wait and retry to get the status

**Ready** – Always returned in the aggregate status file, as VM agent is up and running

**Unresponsive** – Only returned by RDFE when there is no status file uploaded for past 10 mins. Consider this as error state where VM Agent is not behaving correctly

**aggregateStatus\guestAgentStatus\code** - A valid integer status code. Will not be present if code is 0 (meaning healthy)

**aggregateStatus\guestAgentStatus\message** – This is an optional localized message that will be passed back to the user on a GetDeployment call via RDFE.

**aggregateStatus\guestAgentStatus \message\id** – This is the message identifier, to be used for lookup of a localized message. Treated as a string. A symbolic id is preferred for human interpretation, for example Error\_CannotConnect. The file that contains all the localized strings corresponding to the id would be provided by the handler author to Azure during registration.

**aggregateStatus\guestAgentStatus \message\params:** This is an Ordered list of parameter (placeholder) values to be filled into the message template corresponding to the message id. The first Param is used for placeholder “{0}” in the message template (from the provided language resources); the second for placeholder “{1}”, etc.

**aggregateStatus\guestAgentStatus \formattedMessage\lang:** The language/locale of the preformatted message. Like: en-US

**aggregateStatus\guestAgentStatus \formattedMessage\message:** The human readable message that will be returned to the user.

**aggregateStatus\handlerAggregateStatus** – Represents an array of handler’s statuses.

**aggregateStatus\handlerAggregateStatus\handlerName** – Represents the handler full name (Publisher.Name – of the handler) e.g. Microsoft.Compute.CustomScriptExtension

**aggregateStatus\handlerAggregateStatus\handlerVersion** – Represents the version installed on the VM

**aggregateStatus\handlerAggregateStatus\status** – Represents the status of the handler as seen by the VM Agent.

**Installing** – Handler is in process of being installed (user must see the corresponding code, which will have any install specific error code which can be representing VM agent or install script error code – see full list of error codes below)

**NotReady** – Handler is installed but not enabled. (user must see the corresponding code, which will have any enabled specific error code which can be representing VM agent or enable script error code – see full list of error codes below)

**Ready** – Handler is installed and enabled successfully. (user must see message/formatMessage/runtimeSettingsStatus for detailed handler statuses)

**Unresponsive** – Returned if handler opt for heartbeats and did not report any heartbeat status – See section 2.3.1.

**aggregateStatus\handlerAggregateStatus\Code** – Represents error code (0 or not present means no error) corresponding to the status. See some VM Agent specific error codes in the table below.

**aggregateStatus\handlerAggregateStatus\message** & **formattedMessage –** This is optional and will be reported back in following four cases:

Case 1: XML based legacy extension like BGInfo – this will contain VM agent status for install, enable, disable, update status (failure or success)

Case 2: JSON based extension when not opted for heartbeat and does not have any config then this will contain VM agent status for install, enable, disable, update status (failure or success)

Case 3: JSON based extension is not enabled (due to error or is disabled) config then this will contain VM agent status for install, enable, disable, update status (failure or success) or failed to parse status/heartbeat file for the handler

Case 4: JSON based extension’s heartbeat file status message (if opted for heartbeats) – See section 2.3.1 for details.

**aggregateStatus\handlerAggregateStatus\message** – This is an optional localized message that will be passed back to the user on a GetDeployment call via RDFE.

**aggregateStatus\handlerAggregateStatus\message\id** – This is the message identifier, to be used for lookup of a localized message. Treated as a string. A symbolic id is preferred for human interpretation, for example Error\_CannotConnect. The file that contains all the localized strings corresponding to the id would be provided by the handler author to Azure during registration.

**aggregateStatus\handlerAggregateStatus\message\params:** This is an Ordered list of parameter (placeholder) values to be filled into the message template corresponding to the message id. The first Param is used for placeholder “{0}” in the message template (from the provided language resources); the second for placeholder “{1}”, etc.

**aggregateStatus\handlerAggregateStatus\formattedMessage\lang:** The language/locale of the preformatted message. Like: en-US

**aggregateStatus\handlerAggregateStatus\formattedMessage\message:** The human readable message that will be returned to the user.

**aggregateStatus\handlerAggregateStatus\runtimeSettingsStatus –** This is optional and will be reported back only for JSON based extension that processes some config. – See section 2.3.2 for details.

**\*\*\* Caller must wait for one config to be processed before sending a new config for the same handler. VM Agent only sees the last config passed \*\*\***

Caller can look at: **aggregateStatus\handlerAggregateStatus\runtimeSettingsStatus\settingsStatus\timestampUTC** to determine the time when the config was parsed by the handler. IF this time is lower than the time config was pushed then the status retrieved is from the previous config run and caller must retry to get the latest status.

**aggregateStatus\handlerAggregateStatus\runtimeSettingsStatus** – This section represents the handler’s reported status. See section 2.3.2

**aggregateStatus\handlerAggregateStatus\runtimeSettingsStatus\sequencenumber** – Represents the sequence number of the runtime settings / config for which handler is reporting status. \*\*\***This is not exposed to end user via getdeployment call**\*\*\*

**aggregateStatus\handlerAggregateStatus\runtimeSettingsStatus\settingsStatus** – This section will hold all the information reported by that handler. See section 2.3.2 for details

**Sample aggregated status report:**

{

"version": "1.0",

"timestampUTC": "2014-07-16T14:21:31Z",

"aggregateStatus": {

"guestAgentStatus": {

"version": "2.4.1198.689",

"status": "Ready",

"formattedMessage": {

"lang": "en-US",

"message": "GuestAgent is running and accepting new configurations."

}

},

"handlerAggregateStatus": [

{

"handlerName": "Microsoft.Compute.BGInfo",

"handlerVersion": "1.1",

"status": "Ready",

"formattedMessage": {

"lang": "en-US",

"message": "Plugin enabled (name: Microsoft.Compute.BGInfo, version: 1.1)."

}

},

{

"handlerName": "Microsoft.Compute.CustomScriptExtension",

"handlerVersion": "1.0.3",

"status": "Ready",

"runtimeSettingsStatus": {

"sequenceNumber": "0",

"settingsStatus": {

"timestampUTC": "2014-07-12T04:46:13Z",

"status": {

"name": "CustomScriptHandler",

"operation": "Command Execution Finished",

"configurationAppliedTime": "0001-01-01T00:00:00Z",

"status": "success",

"formattedMessage": {

"lang": "en-US",

"message": "Finished executing command"

},

"substatus": [

{

"name": "StdOut",

"status": "success",

"formattedMessage": {

"lang": "en-US",

"message": "output of execution"

}

},

{

"name": "StdErr",

"status": "success",

"formattedMessage": {

"lang": "en-US",

"message": "error output of execution"

}

}

]

}

},

"extensionName": "CustomScriptHandlerConfig1"

}

}

]

}

}

In case VM Agent runs into errors installing, updating, enabling, disabling, uninstalling or status checking then it will report errors it logs and return error code corresponding to each failure.

Here is the list of error codes with the corresponding corrective actions:

|  |  |  |  |
| --- | --- | --- | --- |
| **Error Name** | **ErrorCode** | **Description** | **Corrective Action** |
| PluginUnknownFailure | -1 | VM agent ran into some error  VM agent will retry on next config update or reboot | Failed state |
| PluginSuccess | 0 | No errors | Success |
| PluginProcessingError | 1000 |  |  |
| PluginManifestDownloadError | 1001 | VM agent could not download manifest for the handler. VM agent will retry the operation for 1 hour (max) | Failed state. Caller should wait for max 1 hour and retry |
| PluginHandlerManifestNotFound | 1002 | VM agent could not find the manifest file in handler’s package.  VM agent will retry on next config update or reboot | Failed state |
| PluginHandlerManifestDeserializationError | 1003 | VM agent could not parse the manifest file in handler’s package. VM agent will retry on next config update or reboot | Failed state |
| PluginPackageDownloadFailed | 1004 | VM agent could not download handler’s package. VM agent will retry on next config update or reboot | Failed state |
| PluginPackageExtractionFailed | 1005 | VM agent could not extract handler’s package. VM agent will retry on next config update or reboot | Failed state |
| PluginHandlerFileCreateFailed | 1006 | VM agent could not generate handler environment file. VM agent will retry on next config update or reboot | Failed state |
| PluginInstallProcessingFailed | 1007 | Handler’s install script failed. VM agent will retry on next config update or reboot | Failed state |
| PluginUpdateProcessingFailed | 1008 | Handler’s update script failed. VM agent will retry on next config update or reboot | Failed state |
| PluginEnableProcessingFailed | 1009 | Handler’s enable script failed. VM agent will retry on next config update or reboot | Failed state |
| PluginDisableProcessingFailed | 1010 | Handler’s disable script failed. VM agent will retry on next config update or reboot | Failed state |
| PluginHandlerScriptTimedout | 1011 | Handler’s install/update/enable/disable/uninstall script didn’t complete in time. VM agent will retry on next config update or reboot | Failed state |
| PluginSettingsStatusInvalid | 1012 | VM agent could not parse the handler status file. VM agent will retry in 15-30 seconds | Failed state. Caller should retry for some time, as this can autocorrect. |
| Other non-zero error codes | <non zero error codes> | Any other non-zero error code returned by handler script or config processing by VM Agent | Failed state |

# Logging

Handlers should use the folder provided in the “logfolder” property of the handler environment for writing logs required for debugging their handlers in lieu of any issues reported on a live customer VM.

# Registering via Azure Runtime team.

Currently Azure Runtime team handles most of the registration. The following information needs to be provided by the publisher is Azure Runtime team is registering on behalf of the publisher.

## Subscription for Registration:

The extension should be registered under the Publisher’s Azure Subscription. Prior to Registration, the subscription should be approved for publishing by Azure Runtime team.

If this is register in Pre-Production Environment Azure Runtime team can create the subscriptions.

## Definition file

For registering a handler the following two components are required – the handler package and the definition xml file. The extension handler package needs to be uploaded to a storage location. This section gives an overview of some of the key elements that are required in the definition file.

|  |  |  |
| --- | --- | --- |
| **Property** | **Description** | **Limitations** |
| **ProviderNamespace** | String. This has to be a unique namespace per each subscription. The namespace is a combination of company team, team name (optional) and product name.  E.g.: Microsoft.Azure.RemoteAcccess | Namespace cannot be empty, should be less than 256 chars and underscores cannot be used.  The same namespace cannot be used by multiple subscriptions. |
| **Type** | String. The type of the extension. The type indicate the purpose of the extension. This will be returned as the ‘Type’ property for PaaS List Extension APIs and as ‘Name’ property in IaaS List Extension APIs | Type cannot be empty, should be less than 256 chars and underscores cannot be used. |
| **Version** | String. Version number of the handler. The combination of namespace, type and version uniquely identifies an extension.  The version number needs to be changed for every release. | The format of version number has to be  <major\_version>.<minorver\_version>.<build#>.<revision#>  Eg: 1.0.1.1 |
| **HostingResource** | String. This should be either WebRole or WorkerRole or VmRole depending on whether it’s targeted for PaaS or IaaS. | These values are case sensitive. |
| **MediaLink** | String. URI string pointing to the ZIP package of the extension. The ZIP package has to include the extension executable and other supporting files, and the PluginManifest.xml manifest. | MediaLink value must point to a URL (either Http or Https) in a Windows Azure blob storage and is downloadable. Windows Azure will copy the extension package from this source location and replicate it to all regions. The typical time for replication to complete is 20 minutes. |
| **Certificates** | Certificate thumbprint for encrypting private config | Certificate.StoreLocation must be “LocalMachine”;  Certificate.StoreName must be “My”;  Certificate.ThumbprintAlgorithm must be “sha1”; |
| **PublisherName** | Optional. Publisher text na,e |  |
| **Publisher URL** | A public URL that has usage information and contact information for customer support. |  |
| **EULA** | Optional. If the software requires any additional EULAs, a link to the EULA should be provided. |  |
| **PrivacyUri** | Optional. If the software collects any data and transfers out the VM, then a additional Privacy document might be needed. |  |
| **HomepageUri** | Optional. URL string pointing to homepage of this version of extension. This would be a way for new customers to learn about the extension. |  |
| **Sample Config** | A sample configuration which the user can download and provide his own input parameters. The sample configuration should be sufficiently commented.  If the configuration data is in JSON foarmt, use the following schema for the sample configuration:  {  “Configuration” :  {  “PublicConfig” : <A sample public configuration JSON object>,  “PrivateConfig”: <A sample private configuration JSON object>  }  }  The “PublicConfig” and “PrivateConfig” keys are both optional. |  |
| **IsInternal** | If this is set to ‘true’ the handler is not visible for public use. It can be still accessed by referring to the Namespace, Type & Version combo. | Possible values are case-sensitive true or false |
| **IsJsonExtension** | Boolean string. Optional. If this is set to “true” the extension is expected to receive JSON based configuration. If this is set to “false” the extension is expected to receive XML based configuration. | Case-sensitive true or false. |

**Sample Definition File:**

<?xml version="1.0" encoding="utf-8"?>

<ExtensionImage xmlns="http://schemas.microsoft.com/windowsazure" xmlns:i="http://www.w3.org/2001/XMLSchema-instance">

<ProviderNameSpace>YourCompanyName.YourProductName</ProviderNameSpace>

<Type>YourExtensionName</Type>

<!--Update this version for each new release-->

<Version>1.0</Version>

<Label>Extension Label with Spaces</Label>

<HostingResources>VmRole</HostingResources>

<!--Update this field with correct link where the extension is published.--> <MediaLink>https://Yourprodstorageaccount.blob.core.windows.net/agentextensioncontainer/YourExtensionPackage.zip</MediaLink>

<Certificate />

<Endpoints/>

<PublicConfigurationSchema/>

<PrivateConfigurationSchema/>

<Description>Your Extension Definition</Description>

<PublishedDate>Date Published</PublishedDate>

<LocalResources />

<IsInternalExtension>true</IsInternalExtension>

<SampleConfig>Sample Configuration</SampleConfig>

<Eula>http://yourcompanyEULAUrl/</Eula>

<PrivacyUri>http://yourcompanyPrivacyUrl/</PrivacyUri>

<HomepageUri>http://YourCompanyHomePageWithExtensionInfo</HomepageUri>

<IsJsonExtension>true</IsJsonExtension>

<DisallowMajorVersionUpgrade>false</DisallowMajorVersionUpgrade>

<SupportedOS>Windows | Linux</SupportedOS>

<CompanyName>YourCompanyName</CompanyName>

</ExtensionImage>

# Self-Publishing using APIs

## Register API:

| **Method** | **Request URI** |
| --- | --- |
| POST | https://management.core.windows.net/<subscription-id>/services/extensions |

### Request Body

The description of the tags below can be found the definition section above.

The order among the properties has to match the order in the following table and in the sample request XML.

<ExtensionImage xmlns="http://schemas.microsoft.com/2009/05/WindowsAzure/ServiceManagement" xmlns:i="http://www.w3.org/2001/XMLSchema-instance" >

<ProviderNameSpace>Microsoft.WindowsAzure.Extensions</ProviderNameSpace>

<Type>RDP</Type>

<Version>1.0.0.1</Version>

<Label>extension-label</Label>

<HostingResources>WebRole|WorkerRole</HostingResources>

<MediaLink><https://rdfeagntextacctsn1prod.blob.core.windows.net/AgentExtensionContainer/RemoteAccessPlugin.zip</MediaLink>>

<Certificate>

<StoreLocation>LocalMachine</StoreLocation>

<StoreName>My</StoreName>

<ThumbprintRequired>true</ThumbprintRequired>

<ThumbprintAlgorithm>sha1</ThumbprintAlgorithm>

</Certificate>

<Endpoints>

<InputEndpoints>

<InputEndpoint>

<Name>Microsoft.WindowsAzure.Plugins.RemoteForwarder.RdpInput</Name>

<Protocol>tcp</Protocol>

<Port>3389</Port>

<LocalPort>\*</LocalPort>

</InputEndpoint>

</InputEndpoints>

<InternalEndpoints>

<InternalEndpoint>

<Name>Microsoft.WindowsAzure.Plugins.RemoteAccess.Rdp</Name>

<Protocol>tcp</Protocol>

<Port>3389</Port>

</InternalEndpoint>

</InternalEndpoints>

</Endpoints>

<PublicConfigurationSchema>base-64-encoded-xsd-for-extension-settings</PublicConfigurationSchema>

<PrivateConfigurationSchema>base-64-encoded-xsd-for-extension-settings</PrivateConfigurationSchema>

<Description>detailed extension-description</Description>

<PublisherName>publisher name</PublisherName>

<PublishedDate>2014-02-12T18:55:48Z</PublishedDate>

<LocalResources>

<LocalResource>

<Name>disk space for local caching</Name>

<SizeInMB>50</SizeInMB>

</LocalResource>

</LocalResources>

<BlockRoleUponFailure>false</BlockRoleUponFailure>

<IsInternalExtension>false</IsInternalExtension>

<SampleConfig>base-64-encoded-sample-xml-for-extension</SampleConfig>

<Eula><http://www.contoso.com/42588280809/eula</Eula>>

<PrivacyUri>http://www.contoso.com/42588280809/privacy</PrivacyUri>

<HomepageUri><http://www.contoso.com/42588280809/homepage</HomepageUri>>

<IsJsonExtension>true|false</IsJsonExtension>

</ExtensionImage>

## Update API

| **Method** | **Request URI** |
| --- | --- |
| PUT | https://management.core.windows.net/<subscription-id>/services/extensions?action=update |

The rest body is similar to the ‘Register Extension’ operation.

## Updating Extension

Once the extension is published, any changes to the handler can be published as newer versions, using the update API.

Update is an internal API and will be handled by Azure Runtime until there are public tools for registration.

## Here is an overview of updates are done:

* Hotfixes: Publisher should release hotfixes by changing the revision number. Eg: If the current version is 1.0.0.0, then the hotfixed version would be 1.0.0.1. All hotfixes would be automatically installed on the VM.
* Minor Version Changes: Any minor features can be released as a minor update. E.g.: If the current version is 1.0.0.0, then a minor version update would be 1.1.0.0.

If the client opts in for auto upgrade, all minor version changes would be automatically applied.

* Major Version Change: Any breaking changes in the handler should be released as a major version update. The client has to explicitly request the major version changes.

# Publishing Cmdlets for Handlers

Once the handlers are authored and packages, they can be published using the Azure Extension publishing cmdlets. This is new functionality in addition to publishing via REST APIs.

1. Install the Azure Powershell cmdlets from [here](http://azure.microsoft.com/en-us/documentation/articles/install-configure-powershell/).
2. Open the Azure Powershell cmdlet window and add you Azure Account using Add-AzureAccount.
3. Load the extension publishing module as follows:

PS C:\Users\kundanap\Documents\TechEdDemo> Import-Module "C:\Program Files (x86)\Microsoft SDKs\Azure\PowerShell\ServiceManagement\Azure\Compute\PIR.psd1"

1. To publish a brand new extension, run the following cmdlet:

Publish-AzurePlatformExtension –PublisherNamespace<string> -Type<string> -Version<version#> -Medialink<string url> - Description<String : Extension Description> -PublisherName<String : PublisherName> [-EULA<string : url>] > [-Privacy<string : url>] [-HomePage<string : url>] [-XmlExtension] [-DisallowMajorVersionUpgrade ] [-SampleConfig<string>] [-EndpointConfig<endpoingconfig>] [-CertConfig<certconfig>] [-LocalResourceConfig<localresourceconfig>]

**NOTE: When an extension is published for the first time, it always set as internal.**

1. Once its published, the meta data can be read using the following cmdlet:

Get-AzureVMAvailableExtension -ExtensionName -Publisher

Note : ExtensionName maps to the ‘Type’ in the publisher API and Publisher maps to ‘PublisherNameSpace’

1. Once an extension is published its properties can be updated or the extension handler can be made public using the following cmdlet:

Set-AzurePlatformExtension [-ExtensionName] <string> [-Publisher] <string> [-Version] <string> [[-Label] <string>] [[-Description] <string>] [[-SampleConfig] <string>] [[-Eula] <Uri>] [[-PrivacyUri] <Uri>] [[-HomepageUri] <Uri>] [[-ExtensionMode] <string>] [[-CompanyName] <string>] [-PipelineVariable <string>] [<CommonParameters>]

**NOTE: Once you flip the extension as public using the ExtensionMode Parameter, everybody in the world will see it. So please exert caution.**