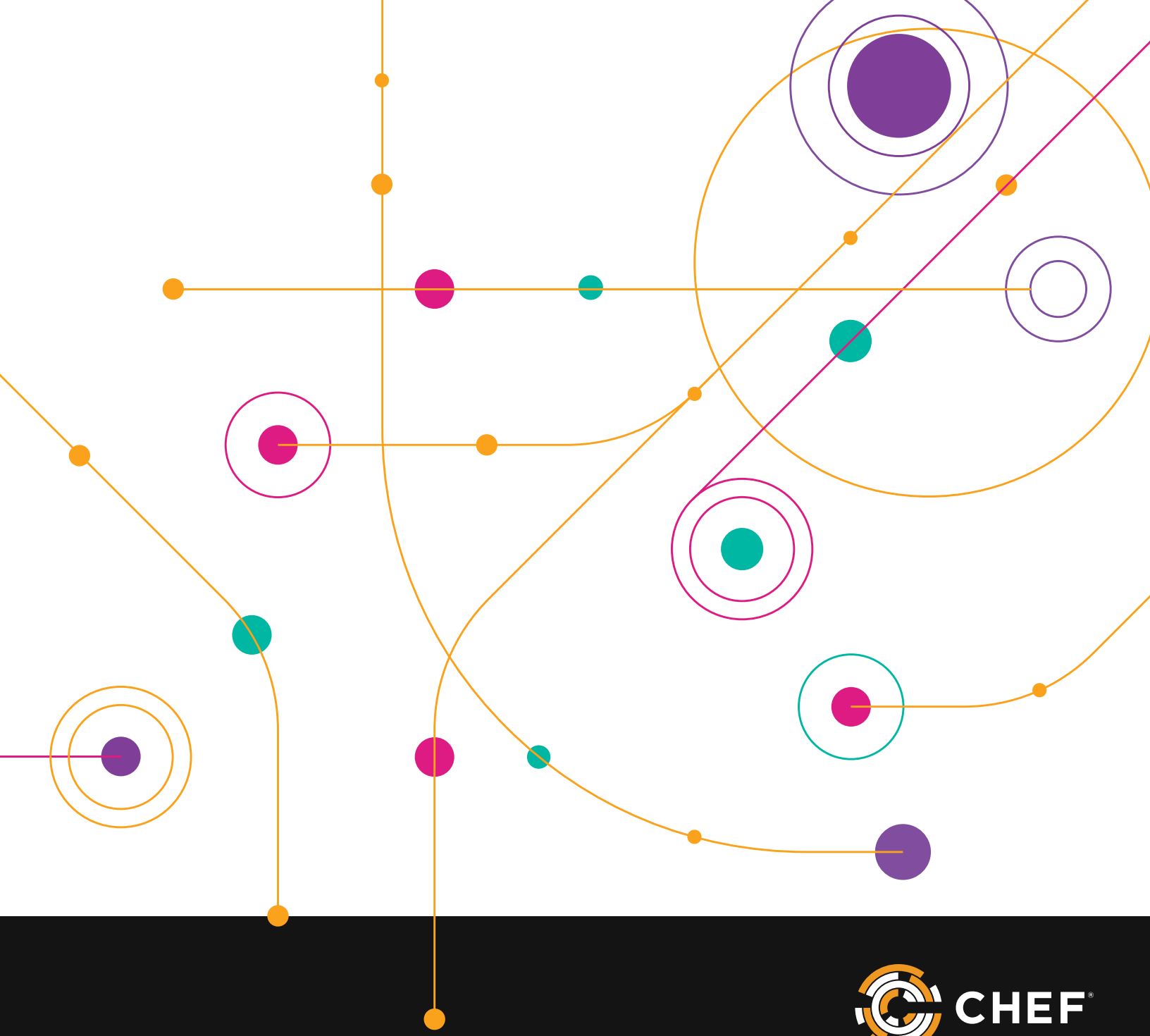

CHEF®

Best Practices for Packaging Windows COTS Applications for Automated Deployment

Application Delivery User Guide

05/2020

Written by: Matt Wrock

Contents

Introduction **1**

Application Compatibility: What COTS Applications can be Packaged with Chef Habitat **1**

1. The Application Must Have a Silent Installer 1

2. The Application Must be Capable of Running on the Target OS 1

Purity vs. Pragmatism: Determining the Right Packaging Approach **1**

The Virtues of Purity: Benefits of Keeping Everything in Chef Habitat 2

When to Take a Pragmatic Approach 3

OS Coupled Applications 3

Isolated Applications 3

Developing a Hook Installed Service Plan **3**

Defining a Portable and Isolatable Plan **3**

Key Considerations: COTS Installers 4

Key Considerations: Breaking Open the Installer **4**

MSI Installers 5

Wix .exe Installers 5

Using 7zip to Extract an .exe Installer 6

Unpacking an Inno Installer 6

When All Options Fail 6

Common Application Install and Configuration Scenarios for Windows COTS Applications **7**

Command Line Executables for Windows COTS Applications 7

Packaging Windows Service Applications 8

Setting Registry Values 9

Website Configuration 10

COM Registration 11

The Build…Fix…Build…Fix…Build…Run Pattern **11**

Read the Logs 11

Look for Missing or Misconfigured Configuration Files 11

Use Process Monitor to Discover Missing Files or the Paths where your Application is Searching for Files 11

Use Process Explorer to Discover the Environment Variables Loaded in a Process 12

Know When to Stop **12**

Introduction

Chef Habitat provides automation capabilities for defining, packaging and delivering applications to almost any environment regardless of operating system or deployment platform.

Chef Habitat application packaging typically falls into one of two main categories:

- Packaging applications built from source code developed in-house or available via open source.
- Packaging commercial off the shelf (COTS) applications usually purchased from a third party vendor.

Habitat plan development patterns and issues to be considered at build and run time vary greatly between these two types of packaging. When it comes to the second class of packages – COTS – the techniques used in Linux are very different from Windows. Linux COTS packaging requires some understanding of ELF headers and patching them to point to Chef Habitat dependencies. Windows binaries do not have ELF headers and use a different means of dynamic linking but there are many subtle and not so subtle nuances involved in packaging COTS applications specific to Windows.

Application Compatibility: What COTS Applications can be Packaged with Chef Habitat

There are two prerequisites that must be met to ensure that a COTS application can run in a Chef Habitat Supervisor.

1. The Application Must Have a Silent Installer

This is almost never a problem. The vast majority of applications are capable of being silently installed. If it does not have a silent installer, you may be able to get by if you choose an approach of packaging without the installer. Read on for more about that.

If you must install via the application installer program and it has no silent install option, there is no automation tool, Chef Habitat included, that can help.

2. The Application Must be Capable of Running on the Target OS

Again, this is usually not a problem. Windows does a great job when it comes to backwards compatibility and many Windows Server 2003 applications will run on Windows Server 2019. However there are absolutely legacy applications that cannot run on modern versions of Windows. Chef Habitat does not employ any extra virtualization/emulation layer to make an application think it is running on an older operating system. For instance, 16 bit applications cannot run on a modern Windows version. There are other applications that may have hard coded checks to ensure they are running on a specific version or within a specific range of versions of Windows. Microsoft SQL Server 2000 is an example of such an application. While it does not happen often, some Windows kernel APIs can change or be removed over time. If an application calls a Win32 API function with an outdated signature, that application may crash on a modern operating system.

Purity vs. Pragmatism: Determining the Right Packaging Approach

When developing a Windows Chef Habitat package for a COTS application, there are typically one of two ways to approach the plan.ps1 and hooks:

1. A Purist Approach

Package runtime artifacts remain in the Chef Habitat file tree and do not affect external Chef Habitat system state. This method of packaging can be challenging but has many benefits.

2. A Pragmatic Approach

The purist path may be too hard, time consuming or down right impossible. Besides, one may not need strict isolation if the Chef Habitat based services are the only applications running on the machine. In such a case, a more pragmatic approach is to just run the application installer binary in the install hook.

There is a bit of a sliding scale between these approaches but the above extremes reflect the spirit of these two packaging methodologies.

Now it may seem like if one can get by on simply packaging the installer and invoking its setup binary at runtime (most likely in an install hook), why go to the trouble to defining a package that pulls out the installer's binaries at build time and reverse engineer any logic the installer might have done at run time? Especially when this logic may be hidden away and almost impossible to reconstruct plainly in Windows COTS installers.

Let's look at why it sometimes makes sense to do the hard thing and why it may make better sense to take the pragmatic route on other occasions.

The Virtues of Purity: Benefits of Keeping Everything in Chef Habitat

Before we get started talking about the benefits Application Definition I just wanted to do a quick recap of the key components of Chef Habitat

- Chef Habitat Studio - A development kit for creating automated build and deployment plans for any application and then testing them in a clean-room environment.
- Chef Habitat Application Artifact (.HART): The Habitat Application Artifact (.HART) is an immutable artifact that is the output of the work done in the Habitat Studio:
- Chef Habitat Supervisor: The Habitat Supervisor is a light-weight agent that runs on/in a server, virtual machine, or container and manages the application according to the instructions defined in the Habitat Plan. Tasks are defined via pre-set scripts called lifecycle hooks that are included as part of the application definition.

Briefly, here are the benefits of the purist approach and taking the time to ensure your application binaries and installation artifacts remain well inside of Habitat boundaries

- Faster service startup. The binaries are already extracted and there is no need to run a possibly lengthy installer process.
- Fewer or no side effects. Loading such a package in the Chef Habitat Supervisor will not contaminate disk space outside of /hab and will possibly not muddy up the registry or the persistable system environment variables.
- Dependencies remain explicit and non magical. It's entirely possible that an installer will install system wide dependencies such as:
  - C runtimes into System32
  - COM component registrations
  - Register .Net assemblies with the Global Assembly Cache (GAC)

Non-Chef Habitat managed services outside of this package may be impacted by these dependencies.

It's especially nice when defining packages in a local Chef Habitat Studio, to know that installing the package I am defining will not have any stateful impact on my machine.

I can blow away the Habitat Studio and my machine remains unscathed. Of course a containerized Studio will avoid this contamination. However, especially when developing COTS packages, troubleshooting in a GUI based environment can be helpful.

Also, updates can be simplified by the purist approach. I don't have to take pains to run an uninstaller or upgrade binary to clean the state of a previous version. My service simply points to the updated /hab/pkgs/ folder.

Now in the Linux world, achieving this kind of purity is much more straightforward. Process state in Linux is very much governed by what is on disk and in environment variables. Windows, on the other hand, has several global system wide APIs that are not easily sandboxed.

When to Take a Pragmatic Approach

There are indeed times when it just is not worth it to take the above purist approach. This is especially true when it could take days to reverse engineer exactly what the installer is doing and you are willing to live with the consequences.

OS Coupled Applications

The Microsoft SQL Server (core/sqlserver) plan is a good example here. The SQL Server installer adds and references many keys into the Windows registry and generates several files including its system database files at install time. Fortunately SQL Server has some isolation capabilities like "named instances" that our plan takes advantage of to limit (but not eliminate) system wide impact.

Isolated Applications

It's also possible that your COTS application may be the only thing ever to run on its node in which case isolation is just a luxury and one you can afford to do without. If you plan to run the COTS application in a containerized Supervisor environment, you automatically get an isolation partition between your application and the host system.

Developing a Hook Installed Service Plan

If you have decided that the best option in your case is to simply install the COTS application using its own installer, here are some guidelines to follow:

- Remember to invoke the installer in the install, init or run hook and not in the plan.ps1 build callback functions. What happens in the build callbacks happens on the build server and not at run time on the Supervisor node. If you are testing in a Studio, these two environments are essentially the same and it can be easy to get fooled into thinking that the application install is working and then wonder why the application is not installed when deploying your package to a Supervisor.
- Be aware of the command line arguments that your application installer accepts. These may give you the option to minimize the impact to the system outside of the Chef Habitat environment. For example, the SQL Server installer has many installation options that control where files are installed. It even allows you to use "Named Instances" that isolate one sql engine from another. This means I can have a named Chef Habitat instance that will leave any existing database on the server alone.

Defining a Portable and Isolatable Plan

Now let us assume you have decided to take the "purist" route described above. The rest of this post will look at the nuts and bolts mechanics of how to actually do that.

Key Considerations: COTS Installers

Before we start breaking the binaries out of the COTS installer, we need to understand what it does. It is easiest to do this on as clean of a system as possible – just a basic OS installation with no other applications. I also suggest doing this on a local hypervisor and snapshot the clean state so that you can easily and quickly restore to a clean state.

Install the application using its own installer. You can do this with its standard GUI wizard or if the application is normally installed from the command line with specific parameters, then install it using that approach. Install it in such a way that best emulates your normal deployment methodology. After the installation look to see how the state of the system has changed. Things to look for include:

- Files in `c:\Program Files`, `c:\Program Files (x86)`
- Windows services in the Services Management Console
- New Environment Variables
- Paths added to the PATH variable
- Web app? – Added sites and pools in IIS
- Added registry keys in HKLM:\SOFTWARE or HKLM:\SOFTWARE\WOW6432Node looking specifically for a subkey named after the vendor

While the above are the most likely system changes you will find, it's also possible you could find these less common changes:

- New tasks in the Windows Task Scheduler
- New COM+ applications
- Newly registered COM components

If you are familiar with the general nature of the application, that will help you to zero in on what to look for. The more you know about the app, the better and more clear the path forward will be. Is this a:

- `*.exe` based tool or service one invokes directly?
- Web Site running on IIS?
- A background Windows Service?
- A *.dll library or COM component (or several) to be used by a top level application?

You will want to make note of any changes you detect after install because your hooks will need to replicate them via Powershell. Also make sure the application actually works. You want to be confident that you are in the desired state that you hope your Chef Habitat plan will reproduce.

Lastly, snapshot or checkpoint your VM here so you can easily and quickly return to a state where the application worked when things go poorly in your Chef Habitat packaging.

Key Considerations: Breaking Open the Installer

Regardless of the type of application being installed, extracting the relevant binaries from the installer file(s) is always going to be the first gating challenge. The method you use to do this can have several permutations depending on the type of installer you have. Unless it is an *.msi based installer, it will not be obvious what type of installer it is.

MSI Installers

Use `core/lessmsi` to extract the files inside of an `.msi` file.

cmake/plan.ps1 link

```
$pkg_source="http://cmake.org/files/v$base_version/cmake-$pkg_version-win64-x64.msi"
$pkg_build_deps=@("core/lessmsi")
function Invoke-Unpack {
  lessmsi x (Resolve-Path "$HAB_CACHE_SRC_PATH/$pkg_filename").Path
  mkdir "$HAB_CACHE_SRC_PATH/$pkg_dirname"
  Move-Item "cmake-$pkg_version-win64-x64/SourceDir/cmake" "$HAB_CACHE_SRC_PATH/$pkg_
dirname"
}
```

`lessmsi` will extract the files into a directory named `SourceDir`. Make sure to use `Resolve-Path` on the msi file in order to send `lessmsi` the absolute and properly Windows formatted path. `lessmsi` does not like forward slashes.

Wix .exe Installers

Most `.exe` based installers (but not all) that come from the Microsoft downloads site seem to be WIX based installers. You can use the `dark.exe` binary in `core/wix` to extract its contents. Often these installers simply contain `msi` files; so you will then need to use `lessmsi` to extract the contents of those.

cmake/plan.ps1 link

```
$pkg_source="https://download.microsoft.com/download/E/F/D/EFD52638-B804-4865-BB57-
47F4B9C80269/NDP462-DevPack-KB3151934-ENU.exe"
$pkg_build_deps=@("core/lessmsi", "core/wix")
$pkg_bin_dirs=@("Program Files\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.2 Tools\x64")
$pkg_lib_dirs=@("Program Files\Windows Kits\NETFXSDK\4.6.2\Lib\um\x64")
$pkg_include_dirs=@("Program Files\Windows Kits\NETFXSDK\4.6.2\Include\um")
function Invoke-Unpack {
  dark -x "$HAB_CACHE_SRC_PATH/$pkg_dirname" "$HAB_CACHE_SRC_PATH/$pkg_filename"
  Push-Location "$HAB_CACHE_SRC_PATH/$pkg_dirname"
  try {
    Get-ChildItem "AttachedContainer/packages" -Include *.msi -Recurse | % {
        lessmsi x $_
    }
  }
  finally { Pop-Location }
}
function Invoke-Install {
  Get-ChildItem "$HAB_CACHE_SRC_PATH/$pkg_dirname" -Include "Program Files" -Recurse | % {
    Copy-Item $_ "$pkg_prefix" -Recurse -Force
  }
}
```

The only sure way to know if an exe is wix based is to use dark to extract its contents. If you get an error, it is not a wix installer.

Using 7zip to Extract an .exe Installer

Some `*.exe` installers can be expanded with core/7zip. Again, it's hard to know if this will work without actually trying it.

jre8/plan.ps1link

```
$pkg_source="http://download.oracle.com/otn-pub/java/jdk/$pkg_upstream_version-b11/
a58eab1ec242421181065cdc37240b08/jre-$pkg_upstream_version-windows-x64.exe"
$pkg_filename="$pkg_name-$pkg_version.exe"
$pkg_build_deps=@("core/7zip")
function Invoke-Unpack() {
    New-Item "$HAB_CACHE_SRC_PATH/$pkg_dirname" -ItemType Directory | Out-Null
    Push-Location "$HAB_CACHE_SRC_PATH/$pkg_dirname"
    try {
        7z x "$HAB_CACHE_SRC_PATH/$pkg_filename"
        7z x data1.cab
        7z x installerexe -ojava
    }
    finally { Pop-Location }
}
```

This `core/jre` exe installer has an embedded `data1.cab` file that contains a `installerexe` file which is what has the actual installer payload.

Unpacking an Inno Installer

Inno is an application setup tool sometimes seen in the COTS world to create `exe` installers. If you open the `.exe` file in a text editor and search the junk text for "Inno," chances are this is an Inno based installer. You can use `core/innounp` to extract an Inno installer.

```
$pkg_build_deps = @("core/innounp")
function Invoke-Unpack {
    mkdir "$HAB_CACHE_SRC_PATH/$pkg_dirname"
    Push-Location "$HAB_CACHE_SRC_PATH/$pkg_dirname"
    try {
        innounp -x (Resolve-Path "$PLAN_CONTEXT/install.exe").Path
    }
  Finally { Pop-Location }
}
```

When All Options Fail

If none of the above options manage to extract your `.exe` installer, it's possible that the `.exe` file can be invoked with the right arguments that will extract its contents without actually performing an application install. The trick is finding out what those args are. I usually try to find any cli help for the `exe` by passing `--help`, `-h`, `/help`, `/h`, `/?`, `-?` to the installer until I get some help. Here is an example of a `exe` that provides an argument to simply extract the contents.

```
Start-Process "$HAB_CACHE_SRC_PATH/$pkg_filename" -Wait -ArgumentList "--passive --layout
$HAB_CACHE_SRC_PATH/$pkg_dirname --lang en-US"
```

This `exe` exposes a `--passive` argument to suppress user interaction and a `--layout` argument to specify a location where the contents should be extracted to ("layed out").

Common Application Install and Configuration Scenarios for Windows COTS Applications

Now that you have managed to pull the raw binaries and other support files out of the installer, what steps do we take to order and further manipulate them so that we have a functioning application managed in Chef Habitat? That is largely going to depend on the type of application you are packaging. In some cases, like with many command line tools and services, this could be super simple – just copy the binaries into the package and add them to `pkg_bin_dirs`. In other cases you may need to add code to your `install`, `init` or `run` hooks to install a Windows Service, set registry keys, and maybe apply some DSC configuration resources. Let's walk through some of the more common scenarios.

Command Line Executables for Windows COTS Applications

These are often the simplest applications to package. They typically just need a `run` hook to invoke the `exe` and possibly pass in some config data or a configuration file. This scenario reflects what we are used to in most Linux based applications and allows us to maintain a purely side-effect free Chef Habitat package. See this Windows based MySQL plan as an example.

If the executable is not a "top level" application but a tool used as a runtime or build dependency, you do not even need hooks. Just make sure the executables are on the path:

node/plan.ps1link

```
$pkg_name="node"
$pkg_origin="core"
$pkg_version="8.11.2"
$pkg_description="Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine."
$pkg_upstream_url="https://nodejs.org/"
$pkg_license=@("MIT")
$pkg_maintainer="The Habitat Maintainers "
$pkg_source="https://nodejs.org/dist/v$pkg_version/node-v$pkg_version-x64.msi"
$pkg_shasum="108dd9ccd131931e42b57dc1243dc28aacfebe207de48c420a576fe453d33ed7"
$pkg_build_deps=@("core/lessmsi")
```

```
$pkg_bin_dirs=@("bin")
function Invoke-Unpack {
lessmsi x (Resolve-Path "$HAB_CACHE_SRC_PATH/$pkg_filename").Path
 mkdir "$HAB_CACHE_SRC_PATH/$pkg_dirname"
  Move-Item "node-v$pkg_version-x64/SourceDir/nodejs" "$HAB_CACHE_SRC_PATH/$pkg_dirname"
}
function Invoke-Install {
  Copy-Item "$HAB_CACHE_SRC_PATH/$pkg_dirname/nodejs/*" "$pkg_prefix/bin" -Recurse
}
```

Here by simply ensuring that `node.exe` is in `pkg_bin_dirs`, other node applications can depend on this and expect invoking node to invoke the executable in this package.

Packaging Windows Service Applications

If the application runs as a Windows Service, you are going to want to `install` the service in your install hook. However the first question to ask is: does this service REALLY need to run as a service? The answer to this question may very well be "no." If it is indeed "no" then the best option is to just run it as a normal executable, which puts us back into the scenario above.

Many Windows service based binaries are capable of running in console mode. That is normally not ideal for a service that you want to run unattended in the background, but that is what the Chef Habitat Supervisor is for. Since the Supervisor can run as a Windows service, you can have it run the executable in console mode but get service-like behavior without creating a system-wide Windows service. This just requires that the service executable can indeed be run directly and that it responds to `ctrl+c` events for stopping the service.

For services that do not meet the above criteria and must be run as a true Windows service, you can add the service in the `install` hook:

hooks/install

```
Set-Location {{pkg.svc_path}}
if(Test-Path bin) { Remove-Item bin -Recurse -Force }
New-Item -Name bin -ItemType Junction -target "{{pkg.path}}/bin" | Out-Null
if((Get-Service my_service -ErrorAction SilentlyContinue) -eq $null) {
    &$env:systemroot\system32\sc.exe create my_service binpath= (Resolve-Path "{{pkg.svc_
path}}\bin\MyService.exe").Path
}
```

It is important that you do not configure the service start mode to `automatic` and keep it at the default `manual` setting because we want the Supervisor to manage its lifecycle.

Rather than directly invoking the executable, your run hook will manage the service `start` and `stop` commands.

hooks/run

```
try {
    Start-Service my_service
    Write-Host "{{pkg.name}} is running"

    while($(Get-Service my_service).Status -eq "Running") {
        Start-Sleep -Seconds 1
    }
}
finally {
    # The service will still be running if the supervisor is stopping the service
    Write-Host "{{pkg.name}} is stopping..."
    if($(Get-Service my_service).Status -ne "Stopped") {
        Stop-Service my_service
    }
    Write-Host "{{pkg.name}} has stopped"
}
```

If your service requires environmental variables to be set, you cannot simply set them in the `run` hook like you could if you were calling the executable directly. Because the Windows service process is not a child process of the hook, you must unfortunately set these environment variables persistently.

hooks/run

```
Write-Host "Setting up environment variables!"
{{#each cfg.env_variables}}
    [Environment]::SetEnvironmentVariable("{{@key}}", "{{this}}", [System.
EnvironmentVariableTarget]::Machine)
{{/each}}
```

Setting Registry Values

Some COTS applications require Windows Registry keys to be created which are later referenced by the application. These are usually located somewhere under `HKLM:\SOFTWARE` under a key named after the software vendor.

If the application just requires a couple keys, it's probably easiest to add them individually in an `install` ,`init` or `run` hook.

hooks/run

```
Set-ItemProperty -Path  "HKLM:\SOFTWARE\Acme\SuperService" -Name DataPath -Value
(Resolve-Path "{{pkg.svc_data_path}}").Path
```

If the values could possibly change upon a `reconfigure`, set them in the `run` hook so that the reconfigured values will update the registry.

If there is a substantial tree of keys and subkeys that the application requires, it's easiest to export the entire tree to a `.reg` file. The file can then be templatized in the `config` directory.

config/tivoli.reg

```
Windows Registry Editor Version 5.00
[HKEY_LOCAL_MACHINE\SOFTWARE\Candle]
[HKEY_LOCAL_MACHINE\SOFTWARE\Candle\KBB_ENVPATH]
"kntcma"="{{pkg.svc_path}}\\bin\\TMAITM~1\\KNTENV"
"kcawd"="{{pkg.svc_path}}\\bin\\TMAITM~1\\KCAENV"
"fcp_daemon"="{{pkg.svc_path}}\\bin\\TMAITM~1\\osfcp\\KFOENV"
"tacmd"="{{pkg.svc_path}}\\bin\\bin\\KUIENV"
"rbaccmd"="{{pkg.svc_path}}\\bin\\bin\\KUIENV"
[HKEY_LOCAL_MACHINE\SOFTWARE\Candle\KNT]
[HKEY_LOCAL_MACHINE\SOFTWARE\Candle\KNT\Ver610]
[HKEY_LOCAL_MACHINE\SOFTWARE\Candle\KNT\Ver610\Primary]
[HKEY_LOCAL_MACHINE\SOFTWARE\Candle\KNT\Ver610\Primary\Configuration]
"Type"="AGENT"
"ProdDesc"="IBM Tivoli Monitoring: Windows OS Agent"
```

The `install`, `init` or `run` hook would then import this template.

hooks/run

```
reg.exe import "{{pkg.svc_config_path}}\tivoli.reg"
```

Website Configuration

If your application runs as a web application, unless it is running an embedded web server, it is likely you will need to configure an IIS Application Pool and Website.

Take a good look at the IIS Management Console where the app is successfully running and observe what pools and sites are dedicated to these apps. For a comprehensive view of these individual entities, look at `C:\Windows\System32\inetsrv\Config\applicationHost.config`. This configuration file will include all of the configuration details without the need for alot of pointing and clicking.

You can express these IIS entities in your plan using Powershell DSC. See this Legacy ASP.NET plan as an example DSC configuration for configuring IIS. To wire up the configuration, make sure to declare a run time dependency on core/dsc-core and invoke its Start-DscCore function in the run hook to actually apply the configuration on the node.

hooks/run

```
Import-Module "{{pkgPathFor "core/dsc-core"}}/Modules/DscCore"
Start-DscCore (Join-Path {{pkg.svc_config_path}} website.ps1) NewWebsite
```

COM Registration

COM registration is becoming less and less frequent but many Legacy Windows applications have COM based dependencies. Especially if multiple applications depend on a COM component, it's best to package that component separately. An example COM based binary in our `core-plans` repository is the sql-dmo package.

This plan's `install` hook registers its DLL so that consuming services are guaranteed that the component is available for use.

hooks/install

```
."$env:SystemRoot\SysWow64\regsvr32.exe" /s "{{pkg.path}}\Program Files (x86)\Microsoft SQL Server\80\Tools\Binn\sqldmo.dll"
```

Note that the above uses the SysWow64 version (32-bit) of the regsvr32 tool which again is common in legacy 32 bit application scenarios. If you were registering the component for a 64-bit application, you would simply call the `regsvr32.exe` directly on the path.

The Build…Fix…Build…Fix…Build…Run Pattern

Especially when you are packaging an application about which you have little knowledge, your initial builds may fail inside of a Supervisor. Sometimes the Supervisor output makes it clear that the service is failing and at other times it may appear as though the hooks executed without error, but the application is not behaving as expected or just isn't running at all.

The path forward in these cases is not always straightforward and we cannot document every edge case here. You will often need to use some creativity and put on your detective hat. Google and StackOverflow may be your best resources, but here are some tips to guide you as you attempt to solve your application packaging dilemmas.

Read the Logs

Which logs? Any logs and all the logs. Look for directories called `log` or `logs`. Check the `system` and `application` Windows Event Logs. These will often have the best clues to help you get to the root of the problem. Sometimes it helps to compare these logs to the logs on a system with a successfully installed application.

Look for Missing or Misconfigured Configuration Files

Some installers may create or write to text files and inject configuration settings like listening port or the location of the installation directory. So look for text files that look like they store key/value pairs. They could be `json`, `xml`, `toml`, `yaml`, or a proprietary format. If you find these and it looks as though their data is incorrect or missing, you may need to add these files to your plan's `config` directory and have the Supervisor inject the correct data at run time and then have your hooks copy or link them to the location where the application expects them to be.

Use Process Monitor to Discover Missing Files or the Paths where your Application is Searching for Files

The application logs may possibly include errors indicating that a file cannot be found, or simply stating that "a file" cannot be found without the courtesy of including the name of the file. You may be missing a dependency or your application is searching the wrong directories for certain files.

Process Monitor (or `procmon`) can be very helpful in solving this problem. It will watch all file activity under a specific process. Just before loading your service in the Supervisor, have `procmon` watch your service process. Stop it right after things fail. Now look for files that it was not able to open. Often you will see it looking in several directories for the same file before it gives up. If you know the file name ahead of time, you can add a `procmon` filter to only include paths that contain the file name.

Maybe you discover that the file it is looking for actually exists in the Chef Habitat package's `svc_path` but the application is looking somewhere else – like `c:\Program Files`. This often indicates an incorrect or missing configuration somewhere. It can be helpful to load the entire application directory into an editor like `vscode` and do a global search for the filename or perhaps `c:\Program Files`. This might surface potential configuration files that you need to templatize in your Chef Habitat plan.

Use Process Explorer to Discover the Environment Variables Loaded in a Process

Process Explorer (procexp) is another tool that can provide some visibility into a process environment. When an application is failing, it may have missing or misconfigured environment variables. Process explorer displays all running processes on a system and allows you to drill into each and observe some basic information about the process. Look at the directory where the process is running from and look at its environment variables. Do you see anything obviously wrong? Do they vary in a meaningful way from the environment variables inside an application that is running correctly?

Know When to Stop

Purity is great but obsession often ends poorly. If you find that deconstructing an installer is sucking up too much time or crushing your soul, strongly consider the "installer in the hook" approach. Remember that in the end, Chef Habitat should be saving you time and trouble and not adding extra burden to your deployments!

Explore Chef Habitat Today!

Chef Habitat Resources and Documentation: https://www.habitat.sh

Chef Free Online Learning: https://learn.chef.io/#/

About the Author: Matt Wrock

Matt is a software engineer on the Chef Habitat team focusing on Windows compatibility and ecosystem development. He has over 20 years of experience as a developer and has made significant contributions to several open source projects. When not cursing at his laptop, Matt enjoys running, reading, piano and walking his dog at the beach.

Chef is the global leader in DevSecOps and the developer of Chef Enterprise Automation Stack™ automating infrastructure, compliance and application delivery for more than half of the Fortune 500. For more than 10 years, Chef has led the industry in DevOps innovation, uniting teams at organizations of all sizes and optimizing processes and outcomes to accelerate its customers' business growth. Chef software is developed as 100 percent open source under the Apache 2.0 license. For more information, visit http://chef.io and follow @chef.