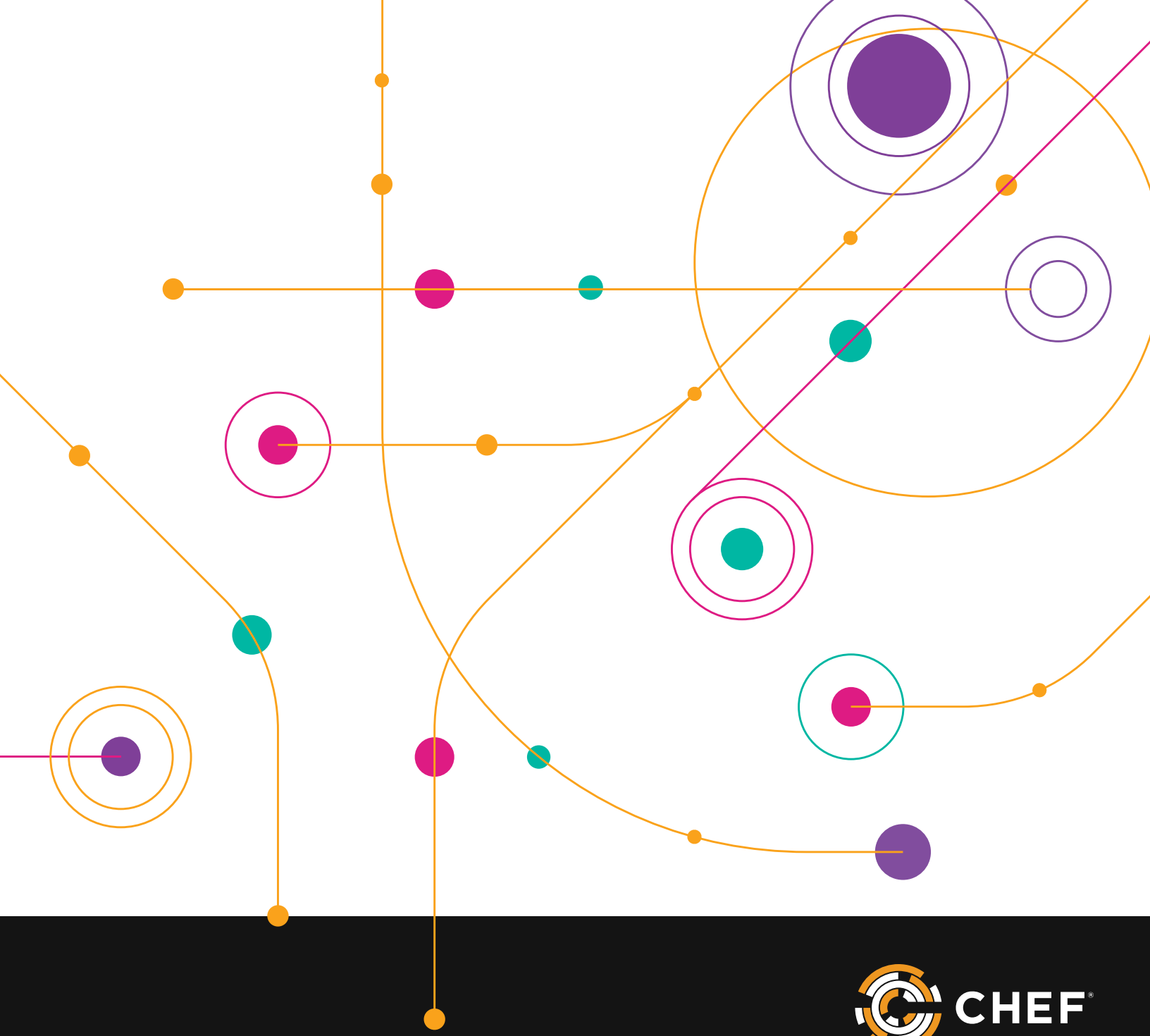

CHEF®

Best Practices for Simplifying and Automating Microsoft SQL Updates

Application Delivery User Guide

05/2020

Written by: Matt Wrock

Contents

Chef Habitat Introduction

Chef Habitat provides automation capabilities for defining, packaging and delivering applications to almost any environment regardless of operating system or deployment platform. The core components of Chef Habitat that we will be touching on in this guide include:

Chef Habitat Studio

The Chef Habitat Studio is a development kit for creating automated build and deployment plans for any application and then testing them in a clean-room environment.

- Define how your application is built, installed, and run with PowerShell or Bash and your code editor of choice.
- Isolate dependencies into atomic plans and then build an Application Manifest which links to all direct & transitive runtime dependencies and provides tuneable instructions to install and run the app.
- Leverage hundreds of pre-built plans for common dependencies such as .NET, jdk or gcc on the Chef Habitat community on GitHub.

Chef Habitat Artifact

Habitat packages the Application Manifest into an immutable artifact called the Habitat Application Artifact (.HART) file. Artifacts can be exported to run in a variety of runtimes with zero refactoring or rewriting.

- Create a package in the local Studio or integrate with any automated pipeline solution using the same commands and build processes.
- Simplify audit and compliance processes by explicitly defining application dependencies and packaging an application with only what is required.
- Easily export to tarball, Docker, or directly to container registries (Azure Container Registry, Amazon Elastic Container Registry, or Docker Hub).

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.

- Deploy and upgrade an application to any environment on-demand
- Automate health and compliance checks
- Templatize your configuration settings and update them as needed during runtime.

As Microsoft SQL is a commercial off the shelf (COTS) software and does not require us to build the application first this guide will mostly deal with how to program the Habitat Supervisor using the built in hooks provided in Chef Habitat. Hooks are used to model deployment tasks as code that can be configured dynamically and executed automatically. Once instructions are defined via a Hook the Chef Habitat Supervisor acts as a workflow engine, coordinating the flow of information between services, and automatically sequencing the events defined in their plans.

Current available hooks include:

- file-updated
- health-check

- init
- reload
- reconfigure
- suitability
- run
- post-run
- post-stop

In addition to providing the means to codify deployment tasks as part of an application artifact, these hooks can make use of Chef Habitat's internal runtime settings. These include parameterized references to system resources (e.g. IP addresses, hostnames), application metadata (e.g. version, dependencies), helper functions (e.g. cluster elections, health status), and user-defined variables & tunables.

Additional information on the supervisor and hooks can be found here:

https://www.habitat.sh/docs/reference/application-lifecycle-hooks/

Microsoft SQL Key Considerations

Microsoft SQL Server has a few characteristics that make it more difficult to package and run than other commercial off the shelf software (COTS).

The SQL Server binaries (like sqlservr.exe and its supporting libraries) cannot simply be extracted and moved to `/hab/pkgs/`. SQL Server makes extensive use of the Windows registry which tracks the location on disk of all SQL Server "instances" along with much of their service metadata. Also, the absolute paths to the data files of the `model` and `msdb` system databases are stored in the `master` database. So in order to reliably install SQL Server in working condition, you need to do so via its installer binaries. Unfortunately this also means you may need to package the installer binaries which weigh in at about 1.5 GB in SQL Server 2017.

Another challenge is that your Chef Habitat `run` hook cannot simply call `sqlservr.exe`. On the surface it seems like that should work since that is what the Windows service invokes and if you run that from a shell, it successfully starts and runs the database engine. However, `sqlservr` does not handle `ctrl+c` signals gracefully, which is how Chef Habitat stops running services by default. The SQL Server engine will prompt you for confirmation to terminate the database and to my knowledge that cannot be suppressed. We could forcibly terminate the process, but that may leave the database in an unrecoverable state. So we do need to have our hooks interact with the SQL Server Windows service.

All of these challenges can be dealt with so let's jump in and explore a working SQL Server Chef Habitat plan. I'll be including the relevant sample Chef Habitat plan and hook code in this post but please see the core-plans repository for a complete Chef Habitat plan.

Getting Started Packaging Microsoft SQL Server with Chef Habitat

SQL Server is a commercial product. You can download an evaluation copy of the database here that will operate for a limited period of time. You can also find a `core/sqlserver` plan in our core-plans repository. This plan will download the free SQL Server Express edition and allow you to specify your own install media in case you want to run your Standard or Enterprise edition. However this plan is best used as a reference for building your own package that either includes or points to your own purchased install media.

For the sake of following along, lets download and extract the free evaluation version of SQL Server 2017:

```
> mkdir download
```

```
> Invoke-WebRequest https://download.microsoft.com/download/5/2/2/522EE642-941E-47A6-8431-57F0C2694EDF/SQLServer2017-SSEI-Eval.exe -OutFile download\sqlsvr.exe
```

```
> .\download\sqlsvr.exe /ACTION=Download /MEDIAPATH=$((Resolve-Path .\download\).path) /MEDIATYPE=Cab /QUIET
```

```
> .\download\SQLServer2017-x64-ENU.exe /x:$((Resolve-Path .\download\).path)\sql /u
```

```
> $env:sqlSetupDir = "$((Resolve-Path .\download\).path)\sql"
```

Note that we set an environment variable `sqlSetupDir` which will be used later to package the install files. Now create a `plan.ps1` file with the contents shown below. You will likely want to use your own `origin` name instead of `mwrock`. This is a plan one might use as an example for their own SQL Server plan. It is extremely simple because it merely copies the install files into the package. It doesn't actually install SQL Server because that will happen in the `install` hook as we will see later.

Here is the plan:

sqlserver/plan.ps1

```
$pkg_name = "sqlserver"
$pkg_origin = "mwrock"
$pkg_version = "0.1.0"
$pkg_maintainer = "The Habitat Maintainers "
$pkg_deps=@("core/dsc-core")
$pkg_exports=@{
    port     ="port"
    password ="app_password"
    username ="app_user"
    instance ="instance"
}
$pkg_description = "Microsoft SQL Server 2017"
$pkg_upstream_url = "https://www.microsoft.com/en-us/sql-server/sql-server-2017"
$pkg_bin_dirs = @("bin")

function Invoke-Install {
```

```
  Copy-Item "$env:sqlSetupDir/*" $pkg_prefix/bin -Recurse
}
```

This assumes that you have the install media located in a directory that is stored in the environment variable named `sqlSetupDir` like we assigned above. Another approach one could take would be to not package the install files at all and instead have the `install` hook point to a network share instead of a huge local payload.

The advantage of storing the install media in the Chef Habitat package is that you are guaranteed that the files needed to install SQL Server travel with the package. However, the package itself will be quite large.

Also notice that we are "exporting" the following configuration properties: port, app_password, app_user, and instance. These are properties that other services can discover about our SQL Server Chef Habitat service at run time and use them to create a connection string and connect to the database.

Installing SQL Server via the `install` Hook

So our plan does not do anything other than "stage" the install files. It's our `install` hook that will do the heavy lifting. It's only going to need to run the installer when the package is installed via `hab pkg install` or the first time we load our Chef Habitat service into a Chef Habitat Supervisor. Be aware that it may take several minutes for the `install` hook to complete.

Our `install` hook will do three things:

1. Install the `xNetworking` Powershell module to be used in our run hook later. Note that we use `Invoke-Command` to wrap the installation of our `xNetworking` DSC module so that it is installed into the Windows Powershell context and not Powershell Core because that is where the DSC Local Configuration Manager runs and uses that module.
2. Install the SqlServer Powershell module to be used later in our `post-run` hook to create a database login.
3. Run the Installer

sqlserver/hooks/install

```
Invoke-Command -ComputerName localhost -EnableNetworkAccess {
    $ProgressPreference="SilentlyContinue"
    Write-Host "Checking for nuget package provider..."
    if(!(Get-PackageProvider -Name nuget -ErrorAction SilentlyContinue -ListAvailable)) {
        Write-Host "Installing Nuget provider..."
        Install-PackageProvider -Name NuGet -Force | Out-Null
    }
    Write-Host "Checking for xNetworking PS module..."
    if(!(Get-Module xNetworking -ListAvailable)) {
        Write-Host "Installing xNetworking PS Module..."
        Install-Module xNetworking -Force | Out-Null
    }
}
```

```
if(!(Get-Module SqlServer -ListAvailable)) {
    Install-Module SqlServer -Force -Scope AllUsers
}



# If the sql instance data is not present, install a new instance
if (!(Test-Path {{pkg.svc_data_path}}/mssql14.{{cfg.instance}})) {
    # Remove any configuration settings set to an empty string
    (Get-Content "{{pkg.svc_config_install_path}}/config.ini" | ? { !$_.EndsWith("``"``"") })
| Set-Content "{{pkg.svc_config_install_path}}/config.ini"
    setup.exe /configurationfile={{pkg.svc_config_install_path}}/config.ini /Q
}
```

Create a `hooks` directory in the same folder where the `plan.ps1` is located and copy the above file contents to `hooks\install`.

If the `install` hook does not find the data files in our Habitat `svc_data_path` then it will run `setup.exe` and use our templatized `config.ini` as the installer inputs. Save this content to `config_install\config.ini`.

sqlserver/hooks/install

```
[OPTIONS]
ACTION="Install"
IACCEPTSQLSERVERLICENSETERMS=
UpdateEnabled="0"
FEATURES="{{cfg.features}}"
INSTANCEID="{{cfg.instance}}"
INSTANCENAME="{{cfg.instance}}"
AGTSVCSTARTUPTYPE="Manual"
INSTALLSQLDATADIR="{{pkg.svc_data_path}}"
SQLSYSADMINACCOUNTS="{{cfg.sys_admin_account}}"
SQLSVCACCOUNT="{{cfg.svc_account}}"
SQLSVCPASSWORD="{{cfg.svc_account_password}}"
SQLSVCSTARTUPTYPE="Manual"
SQLBACKUPDIR="{{pkg.svc_data_path}}\backup"
SQLTEMPDBDIR="{{pkg.svc_data_path}}"
SQLTEMPDBLOGDIR="{{pkg.svc_data_path}}"
SQLUSERDBDIR="{{pkg.svc_data_path}}"
SQLUSERDBLOGDIR="{{pkg.svc_data_path}}"
TCPENABLED="1"
SECURITYMODE="SQL"
SAPWD="{{cfg.sa_password}}"
```

Many of the values we use here we want to be configurable on a per environment basis. We will include default values in our default.toml file:

sqlserver/hooks/install

```
sa_password="Pass@word1"
port=8888
app_user="hab"
app_password="h@b1Tat"
instance="hab_sql_server"
sys_admin_account="Administrator"
svc_account="NT AUTHORITY\\Network Service"
svc_account_password=""
features="SQLEngine"
```

This file should be saved in the root of our plan directory along with the `plan.ps1`.

We only install the SQLEngine feature assuming we just need basic database services and no reporting, OLAP, or other fancy stuff. We also set the `INSTANCEID` and `INSTANCENAME` to match our package name as well as place all SQL data files in the `svc_data_path` of our Chef Habitat service. This grants us some portability. If we were to deploy this to a machine that already had SQL Server installed, this adds another named instance unique to our package name.

Note that in many large database production environments, you may want more control over where the data files are stored. It might not be feasible to place them all in the service's data path. If that is the case, you could use something like `{{cfg.data_dir}}` instead of `{{pkg.svc_data_path}}`. The Chef Habitat Supervisor decides where `{{pkg.svc_data_path}}` resides, but you can configure a `data_dir` property that you can set a default value for in a default.toml file (see later in this article for an example `default.toml`) and override in different environments at run time.

Setting the Port and Opening the Firewall

By default SQL Server listens on port 1433 and then forwards requests to a dynamically allocated port for the targeted named instance. This assumes the `SQLBrowser` service is running, but we don't want to depend on that if we can avoid it since Habitat is not running it. Instead we can assign our instance a static port and then our applications can send requests to that port. One typically assigns static ports using the SQL Configuration Manager GUI, but we can accomplish the same end by setting a registry key. Add the following to a file named `run` in our `hooks` directory.

sqlserver/hooks/run

```
# Configure the instance for the configured port
Set-ItemProperty -Path  "HKLM:\SOFTWARE\Microsoft\Microsoft SQL Server\MSSQL14.{{cfg.
instance}}\MSSQLServer\SuperSocketNetLib\Tcp\IPAll" -Name TcpPort -Value {{cfg.port}}
```

We will use DSC (Desired State Configuration) to ensure our local firewall allows inbound traffic on that port. We define our DSC configuration in `config/firewall.ps1`:

sqlserver/config/firewall.ps1

```
Configuration NewFirewallRule
{
    Import-DscResource -Module xNetworking
    Node 'localhost' {
        xFirewall "sqlserver-{{pkg.name}}"
        {
            Name    = "sqlserver-{{pkg.name}}"
            DisplayName = "sqlserver-{{pkg.name}}"
            Action = "Allow"
            Direction = "InBound"
            LocalPort = ("{{cfg.port}}")
            Protocol = "TCP"
            Ensure = "Present"
            Enabled  = "True"
        }
    }
}
```

And our `run` hook invokes this. So add the following to the `run` hook:

habitat-sql-server/hooks/run

```
if($(Get-Service 'MpsSvc').Status -eq "Running") {
    Import-Module "{{pkgPathFor "core/dsc-core"}}/Modules/DscCore"
    Start-DscCore (Join-Path {{pkg.svc_config_path}} firewall.ps1) NewFirewallRule
}
```

This leverages our `core/dsc` Habitat package which we declared a dependency for in our `plan.ps1` above and makes it easy to apply a DSC configuration in Chef Habitat's Powershell core environment. Also note that we only configure the firewall rule if the firewall service (MpsSvc) is actually running. For example a container environment will not be running a firewall.

Why did we put the firewall setup in the `run` hook and not the `install` hook? The `install` hook only runs the very first time our package is installed on a machine. We may decide that we want to change the port that our instance listens on after installation. The `run` hook is run every time the Chef Habitat Supervisor loads or starts the service. It is also executed any time we change runtime configuration values at run time.

Running the SQL Server Service

Our `run` hook is going to start the `MSSQL` service installed for our SQL named instance and then spin until the service is stopped. Following is the complete contents of the run hook including the port configuration and the logic to run the service:

sqlserver/hooks/run

```
# Configure the instance for the configured port
Set-ItemProperty -Path  "HKLM:\SOFTWARE\Microsoft\Microsoft SQL Server\MSSQL14.{{cfg.
instance}}\MSSQLServer\SuperSocketNetLib\Tcp\IPAll" -Name TcpPort -Value {{cfg.port}}

if($(Get-Service 'MpsSvc').Status -eq "Running") {
    Import-Module "{{pkgPathFor "core/dsc-core"}}/Modules/DscCore"
    Start-DscCore (Join-Path {{pkg.svc_config_path}} firewall.ps1) NewFirewallRule
}

Start-Service 'MSSQL${{cfg.instance}}'
Write-Host "{{pkg.name}} is running"

try {
    while($(Get-Service 'MSSQL${{cfg.instance}}').Status -eq "Running") {
        Start-Sleep -Seconds 1
    }
}
finally {
    $currentStatus = (Get-Service 'MSSQL${{cfg.instance}}').Status
    if($currentStatus -eq "Running") {
        Write-Host "{{pkg.name}} stopping..."
        Stop-Service 'MSSQL${{cfg.instance}}'
        $currentStatus = (Get-Service 'MSSQL${{cfg.instance}}').Status
    }
    if($currentStatus -eq "StopPending") {
        Write-Host "Waiting for {{pkg.name}} to stop..."
        while($currentStatus -eq "StopPending") {
            Start-Sleep -Seconds 5
            $currentStatus = (Get-Service 'MSSQL${{cfg.instance}}').Status
        }
    }
    Write-Host "{{pkg.name}} has stopped"
}
```

Note that the service name will always be suffixed with our instance name which we have set to be equal to our package name.

This is a standard Chef Habitat best practice for wrapping a Windows service. Instead of invoking an application binary, we start the Windows service. We then simply spin in a loop because if we didn't, our hook would exit which would communicate to the Chef Habitat Supervisor that the application has ended, potentially causing the Supervisor to attempt to restart the service. We want the `run` hook to continue running for as long as the service (SQL Server in our case) is running. If something were to cause the SQL Server service to terminate, then our `if` condition would no longer

be true and our hook would terminate.

Furthermore, if we intentionally tell the Supervisor to stop the service, the Supervisor will send a `ctrl+c` signal to the hook causing it to exit the while loop and enter the `finally` block. This `finally` block checks to see if the service is still running and if so, cleanly stops the service.

Setting up Logins and Users

After Sql Server is installed we have an all powerful `sa` (System Administrator) user and the local `Administrator` Windows user is designated an admin user in our config.ini. We don't want our application to access the database as the `sa` user. We'll add a `post-run` hook that will run after Habitat starts our `sqlserver` service and it will ensure an application user and password are configured:

sqlserver/hooks/post-run

```
# Create application Users

if('{{cfg.app_user}}' -ne '') {
  $listening = $false
  while(!$listening) {
    Write-Host "waiting for sql server to start accepting connections..."
    Start-Sleep -Seconds 3
    try{
      $listening = New-Object System.Net.Sockets.TCPClient -ArgumentList localhost,{{cfg.
port}} -ErrorAction SilentlyContinue | ? { $_.Connected }
    } catch {}
  }

  $saPassword = ConvertTo-SecureString '{{cfg.sa_password}}' -AsPlainText -Force
  $saCred = New-Object System.Management.Automation.PSCredential ('sa', $saPassword)
  $instance = "{{cfg.instance}}"

  if(!(Get-SqlLogin -LoginName {{cfg.app_user}} -Credential $saCred -ServerInstance
"localhost\$instance" -ErrorAction SilentlyContinue)) {
    Write-Host "Starting application user setup..."
    if('{{cfg.app_password}}' -eq '') {
      add-SqlLogin -LoginName {{cfg.app_user}} -Credential $saCred -ServerInstance
"localhost\$instance" -LoginType WindowsUser -Enable -GrantConnectSql
    } else {
      $loginPassword = ConvertTo-SecureString '{{cfg.app_password}}' -AsPlainText -Force
      $loginCred = New-Object System.Management.Automation.PSCredential ('{{cfg.app_
user}}', $loginPassword)
      add-SqlLogin -LoginName {{cfg.app_user}} -Credential $saCred -ServerInstance
"localhost\$instance" -LoginType SqlLogin -LoginPSCredential $loginCred -Enable
-GrantConnectSql
```

```
    }
    Invoke-Sqlcmd -Credential $saCred -ServerInstance "localhost\$instance" -Query
"create user [{{cfg.app_user}}] for login [{{cfg.app_user}}]"
    Write-Host "Application user setup complete"
  }
}
```

This waits to make sure SQL Server is accepting connections and then uses the `SqlServer` Powershell module to execute commands that creates an application login and sets up that login as a user of the `master` database. Note that if we did not configure an application user password, this hook assumes that we want to create a Windows user login instead of a SQL login.

The key thing to understand about the post-run hook is that is where any configuration operations can be declared to be performed after a service is running. For SQL Server this could include configuring backups, creating initial database schema, attaching data files, etc.

Exporting SQL Server to a Docker Image

While you may not be planning to run SQL Server inside containers in a production environment, it can be quite handy to spin up a SQL Server container for application development without needing to install a specific version of SQL Server on my local system especially if I already have a different version of SQL Server running.

Because our Chef Habitat plan includes everything needed for the Supervisor to install, configure and run SQL Server, it provides all the necessary ingredients needed to create a dockerfile and build a Docker image. That is exactly what the `hab pkg export docker` command does. The command takes the package identifier of any Chef Habitat service (`mwrock/sqlserver` in my case) as an argument and builds an image that can run that service.

There are a couple things we need to consider when building an image for SQL Server. First, we want to specify NT `AUTHORITY\SYSTEM` as the service account running SQL Server. We will run into problems if we stop our container and restart it using our default service account of `NT AUTHORITY\Network Service`. Lastly, if we are building the image on Windows 10 or using Hyper-V isolation, we need to specify at least 2 GB of memory to be used as opposed to the default 1 GB allocated to the container VM. This value can be passed to `hab pkg export docker`.

Before we build our image, let's first just make sure our plan directory structure is as expected. From the root of the plan directory, your directory tree should look like this:

```
|  default.toml
|  plan.ps1
|
|---config
|     firewall.ps1
|
|---config_install
```

```
|     config.ini
|
|---hooks
      install
      post-run
      run
```

So let's build our Chef Habitat package and then export it to a Docker image. We want to make sure that we are in the root of our Chef Habitat plan directory when we run this.

```
hab pkg build .
. .\results\last_build.ps1
$env:HAB_SQLSERVER="{`"svc_account`":`"NT AUTHORITY\\SYSTEM`"}"
hab pkg export docker --memory 2gb .\results\$pkg_artifact
```

This builds the package artifact (.hart file). Then we set the `HAB_SQLSERVER` environment variable to a JSON snippet used to override values in our package's `default.toml` file. This environment variable follows a Chef Habitat convention where we can override configuration using a variable prefixed with `HAB_` and adding the name of our service. Then the last line builds the Docker image specifying that we use 2GB (because I am building this on Windows 10 which would default to 1GB leading to a failed SQL Server installation) and point to the path of our package artifact file. Note that because we dot sourced `.\results\last_build.ps1` produced by the package build, the `$pkg_artifact` is populated for us.

When Chef Habitat builds our image, it includes the Chef Habitat Supervisor as well as our service and all of its dependencies. It also runs the `install` hooks of the service and its dependencies. This means that when our image is built, it has a complete SQL Server installation which means that the containers we start from this image should start quickly.

Starting the SQL Server container is now just a matter of calling `docker run` with the built image which is named after our origin/service name.

```
docker run -it --env HAB_LICENSE=accept-no-persist -p 8888:8888 $pkg_origin/$pkg_name
```

This runs the container interactively so we can see the Chef Habitat Supervisor log on the console. It also sets the Chef Habitat license environment variable required for the Supervisor to run and publishes port 8888 which is what our default.toml file specifies as the port SQL Server will listen on.

Testing Connectivity to a "Habitized" SQL Server

Lets see if we can connect to our database container using our application user defined in our `default.toml:`

To test this I am going to install the `SqlServer` powershell module allowing me to issue arbitrary SQL queries using the `Invoke-Sqlcmd` cmdlet.

```
Install-Module SqlServer -Force
```

```
$p = ConvertTo-SecureString 'h@b1Tat' -AsPlainText -Force
```

```
$c = New-Object System.Management.Automation.PSCredential ('hab', $p)
```

```
Invoke-Sqlcmd -ServerInstance "localhost,8888" -Credential $c -Query "select name from sys.databases"
```

I should now see a list of the system databases installed with SQL Server!

Using a Habitat Update strategy to patch SQL Server

Now let's take a look at how we can use Chef Habitat update strategies to automate patching and updating our SQL Server service. First we will locally download the latest SQL Server 2017 cumulative update:

```
> mkdir c:\cu
```

```
> Invoke-WebRequest https://download.microsoft.com/download/C/4/F/C4F908C9-98ED-4E5F-88D5-7D6A5004AEBD/SQLServer2017-KB4541283-x64.exe -OutFile c:\cu\update.exe
```

We should also make sure to update the version of our package to match the version of the cumulative update in our `plan.ps1`:

```
$pkg_version = "14.0.3294"
```

Now let's add the following to the end of our `install` hook:

```
$sqlexe = Get-Item "$env:ProgramFiles\Microsoft SQL Server\MSSQL14.{{cfg.instance}}\MSSQL\Binn\sqlservr.exe" -ErrorAction SilentlyContinue
```

```
if($sqlexe) {
```

```
    if([Version]::new($sqlexe.VersionInfo.ProductVersion) -lt [Version]::new('{{pkg.version}}')) {
```

```
        Write-Host "Patching to {{pkg.version}}..."
```

```
        Start-Process {{cfg.cu_media_dir}}\update.exe -ArgumentList "/q /IAcceptSQLServerLicenseTerms /Action=Patch /AllInstances" -Wait
```

```
    }
```

```
}
```

This will compare the version of the installed sqlservr.exe binary to our package plan version. If SQL Server is at a lesser version, the hook will perform the patching. It will expect to find the patch from a configurable location ({{cfg.cu_media_dir}}). We will add this to our default.toml file and point it to where we downloaded the update:

```
cu_media_dir="C:/cu"
```

For the purposes of this walk through we will not include the update binary inside of our build package artifact. Packaging both the complete SQL Server install media and the update would build an artifact larger than 2 gigabytes. Iterating over builds and uploads to test our update with such a large package could become very time consuming.

Now we will start a supervisor loading our sqlserver 0.1.0 package and configure it to use an `at-once` update strategy listening to a `sql_update` channel.

```
> Hab sup run .\results\$pkg_artifact --strategy at-once --channel sql_update
```

This loads our initial sqlserver plan into a local supervisor. The service is loaded with an at-once strategy listening to the sql_update channel for updates. This means that if we promote an updated package to the sql_update channel, our supervisor should update the package and we should see the instance perform a patch.

So let's build our new package with the update and upload it to the Chef Habitat depot:

```
> hab pkg build .
```

```
> . .\results\last_build.ps1
```

```
> hab pkg upload mwrock/$pkg_artifact
```

Now let's promote the package to our `sql_update` channel:

```
> hab pkg promote mwrock/sqlserver/14.0.3294/20200518171319 sql_update
```

Within a minute you should notice the Chef Habitat supervisor output indicating that it is loading the new version and patching to 14.0.3294.

When the patching is complete, you can confirm with:

```
> (Get-Item "$env:ProgramFiles\Microsoft SQL Server\MSSQL14.HAB_SQL_SERVER\MSSQL\Binn\
sqlservr.exe").VersionInfo.ProductVersion
```

This should return 14.0.3294.2.

Exploring a plan for SQL Server High Availability

We have shown here how to use Chef Habitat to deploy a SQL Server service and how to manage SQL Server updates. Of course when SQL Server updates there will be some downtime incurred because the patching process will stop and update the SQL Server binaries. Using a SQL Server high availability solution like Always On Availability Groups, one could run a pair (or more) of SQL Server instances wherein a secondary instance can be updated while your primary instance is serving sql traffic. Once updated the primary instance can fail over to the secondary and then patch itself resulting in the pair being updated with no down time to connecting applications.

A walk through of building and deploying SQL Server instances in a highly available scenario is beyond the scope of this guide. However, you may visit https://github.com/habitat-sh/sql_server_poc to get an idea of what would go into such a plan. The repository includes terraform scripts that can build the VMs, network and load balancer needed to get SQL Server Availability Groups working in an Azure cloud environment. The sqlserver-ha plan includes all of the hook logic needed to setup the Windows Failover Clustering, Availability Group components and database backup and restore for the secondary replicas.

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.