Windows module development walkthrough

In this section, we will walk through developing, testing, and debugging an Ansible Windows module.

Because Windows modules are written in Powershell and need to be run on a Windows host, this guide differs from the usual development walkthrough guide.

What's covered in this section:

Windows environment setup

Unlike Python module development which can be run on the host that runs Ansible, Windows modules need to be written and tested for Windows hosts. While evaluation editions of Windows can be downloaded from Microsoft, these images are usually not ready to be used by Ansible without further modification. The easiest way to set up a Windows host so that it is ready to by used by Ansible is to set up a virtual machine using Vagrant. Vagrant can be used to download existing OS images called *boxes* that are then deployed to a hypervisor like VirtualBox. These boxes can either be created and stored offline or they can be downloaded from a central repository called Vagrant Cloud.

This guide will use the Vagrant boxes created by the packer-windoze repository which have also been uploaded to Vagrant Cloud. To find out more info on how these images are created, please go to the GitHub repo and look at the `README` file.

Before you can get started, the following programs must be installed (please consult the Vagrant and VirtualBox documentation for installation instructions):

- Vagrant
- VirtualBox

Create a Windows server in a VM

To create a single Windows Server 2016 instance, run the following:

```
vagrant init jborean93/WindowsServer2016
vagrant up
```

This will download the Vagrant box from Vagrant Cloud and add it to the local boxes on your host and then start up that instance in VirtualBox. When starting for the first time, the Windows VM will run through the sysprep process and then create a HTTP and HTTPS WinRM listener automatically. Vagrant will finish its process once the listeners are online, after which the VM can be used by Ansible.

Create an Ansible inventory

The following Ansible inventory file can be used to connect to the newly created Windows VM:

```
[windows]
WindowsServer  ansible_host=127.0.0.1

[windows:vars]
ansible_user=vagrant
ansible_password=vagrant
ansible_port=55986
ansible_connection=winrm
ansible_winrm_transport=ntlm
ansible_winrm_server_cert_validation=ignore
```

> **Note**
>
> The port `55986` is automatically forwarded by Vagrant to the Windows host that was created, if this conflicts with an existing local port then Vagrant will automatically use another one at random and display show that in the output.

The OS that is created is based on the image set. The following images can be used:

- jborean93/WindowsServer2012
- jborean93/WindowsServer2012R2
- jborean93/WindowsServer2016
- jborean93/WindowsServer2019
- jborean93/WindowsServer2022

When the host is online, it can accessible by RDP on `127.0.0.1:3389` but the port may differ depending if there was a conflict. To get rid of the host, run `vagrant destroy --force` and Vagrant will automatically remove the VM and any other files associated with that VM.

While this is useful when testing modules on a single Windows instance, these host won't work without modification with domain based modules. The Vagrantfile at ansible-windows can be used to create a test domain environment to be used in Ansible. This repo contains three files which are used by both Ansible and Vagrant to create multiple Windows hosts in a domain environment. These files are:

- `Vagrantfile`: The Vagrant file that reads the inventory setup of `inventory.yml` and provisions the hosts that are required
- `inventory.yml`: Contains the hosts that are required and other connection information such as IP addresses and forwarded ports
- `main.yml`: Ansible playbook called by Vagrant to provision the domain controller and join the child hosts to the domain

By default, these files will create the following environment:

- A single domain controller running on Windows Server 2016
- Five child hosts for each major Windows Server version joined to that domain
- A domain with the DNS name `domain.local`
- A local administrator account on each host with the username `vagrant` and password `vagrant`
- A domain admin account `vagrant-domain@domain.local` with the password `VagrantPass1`

The domain name and accounts can be modified by changing the variables `domain_*` in the `inventory.yml` file if it is required. The inventory file can also be modified to provision more or less servers by changing the hosts that are defined under the `domain_children` key. The host variable `ansible_host` is the private IP that will be assigned to the VirtualBox host only network adapter while `vagrant_box` is the box that will be used to create the VM.

Provisioning the environment

To provision the environment as is, run the following:

```
git clone https://github.com/jborean93/ansible-windows.git
cd vagrant
vagrant up
```

> **Note**
>
> Vagrant provisions each host sequentially so this can take some time to complete. If any errors occur during the Ansible phase of setting up the domain, run `vagrant provision` to rerun just that step.

Unlike setting up a single Windows instance with Vagrant, these hosts can also be accessed using the IP address directly as well as through the forwarded ports. It is easier to access it over the host only network adapter as the normal protocol ports are used, for example RDP is still over `3389`. In cases where the host cannot be resolved using the host only network IP, the following protocols can be access over `127.0.0.1` using these forwarded ports:

- `RDP`: 295xx
- `SSH`: 296xx
- `WinRM HTTP`: 297xx
- `WinRM HTTPS`: 298xx
- `SMB`: 299xx

Replace `xx` with the entry number in the inventory file where the domain controller started with `00` and is incremented from there. For example, in the default `inventory.yml` file, WinRM over HTTPS for `SERVER2012R2` is forwarded over port `29804` as it's the fourth entry in `domain_children`.

Windows new module development

When creating a new module there are a few things to keep in mind:

- Module code is in Powershell (.ps1) files while the documentation is contained in Python (.py) files of the same name
- Avoid using `Write-Host/Debug/Verbose/Error` in the module and add what needs to be returned to the `$module.Result` variable
- To fail a module, call `$module.FailJson("failure message here")`, an Exception or ErrorRecord can be set to the second argument for a more descriptive error message
- You can pass in the exception or ErrorRecord as a second argument to `FailJson("failure", $_)` to get a more detailed output
- Most new modules require check mode and integration tests before they are merged into the main Ansible codebase
- Avoid using try/catch statements over a large code block, rather use them for individual calls so the error message can be more

descriptive
- Try and catch specific exceptions when using try/catch statements
- Avoid using PSCustomObjects unless necessary
- Look for common functions in `./lib/ansible/module_utils/powershell/` and use the code there instead of duplicating work. These can be imported by adding the line `#Requires -Module *` where * is the filename to import, and will be automatically included with the module code sent to the Windows target when run via Ansible
- As well as PowerShell module utils, C# module utils are stored in `./lib/ansible/module_utils/csharp/` and are automatically imported in a module execution if the line `#AnsibleRequires -CSharpUtil *` is present
- C# and PowerShell module utils achieve the same goal but C# allows a developer to implement low level tasks, such as calling the Win32 API, and can be faster in some cases
- Ensure the code runs under Powershell v3 and higher on Windows Server 2012 and higher; if higher minimum Powershell or OS versions are required, ensure the documentation reflects this clearly
- Ansible runs modules under strictmode version 2.0. Be sure to test with that enabled by putting `Set-StrictMode -Version 2.0` at the top of your dev script
- Favor native Powershell cmdlets over executable calls if possible
- Use the full cmdlet name instead of aliases, for example `Remove-Item` over `rm`
- Use named parameters with cmdlets, for example `Remove-Item -Path C:\temp` over `Remove-Item C:\temp`

A very basic Powershell module win_environment incorporates best practices for Powershell modules. It demonstrates how to implement check-mode and diff-support, and also shows a warning to the user when a specific condition is met.

A slightly more advanced module is win_uri which additionally shows how to use different parameter types (bool, str, int, list, dict, path) and a selection of choices for parameters, how to fail a module and how to handle exceptions.

As part of the new `AnsibleModule` wrapper, the input parameters are defined and validated based on an argument spec. The following options can be set at the root level of the argument spec:

- `mutually_exclusive`: A list of lists, where the inner list contains module options that cannot be set together
- `no_log`: Stops the module from emitting any logs to the Windows Event log
- `options`: A dictionary where the key is the module option and the value is the spec for that option
- `required_by`: A dictionary where the option(s) specified by the value must be set if the option specified by the key is also set
- `required_if`: A list of lists where the inner list contains 3 or 4 elements;
    - The first element is the module option to check the value against
    - The second element is the value of the option specified by the first element, if matched then the required if check is run
    - The third element is a list of required module options when the above is matched
    - An optional fourth element is a boolean that states whether all module options in the third elements are required (default: `$false`) or only one (`$true`)
- `required_one_of`: A list of lists, where the inner list contains module options where at least one must be set
- `required_together`: A list of lists, where the inner list contains module options that must be set together
- `supports_check_mode`: Whether the module supports check mode, by default this is `$false`

The actual input options for a module are set within the `options` value as a dictionary. The keys of this dictionary are the module option names while the values are the spec of that module option. Each spec can have the following options set:

- `aliases`: A list of aliases for the module option
- `choices`: A list of valid values for the module option, if `type=list` then each list value is validated against the choices and not the list itself
- `default`: The default value for the module option if not set
- `deprecated_aliases`: A list of hashtables that define aliases that are deprecated and the versions they will be removed in. Each entry must contain the keys `name` and `collection_name` with either `version` or `date`
- `elements`: When `type=list`, this sets the type of each list value, the values are the same as `type`
- `no_log`: Will sanitise the input value before being returned in the `module_invocation` return value
- `removed_in_version`: States when a deprecated module option is to be removed, a warning is displayed to the end user if set
- `removed_at_date`: States the date (YYYY-MM-DD) when a deprecated module option will be removed, a warning is displayed to the end user if set
- `removed_from_collection`: States from which collection the deprecated module option will be removed; must be specified if one of `removed_in_version` and `removed_at_date` is specified
- `required`: Will fail when the module option is not set
- `type`: The type of the module option, if not set then it defaults to `str`. The valid types are;
    - `bool`: A boolean value
    - `dict`: A dictionary value, if the input is a JSON or key=value string then it is converted to dictionary
    - `float`: A float or Single value
    - `int`: An Int32 value
    - `json`: A string where the value is converted to a JSON string if the input is a dictionary
    - `list`: A list of values, `elements=<type>` can convert the individual list value types if set. If `elements=dict` then `options` is defined, the values will be validated against the argument spec. When the input is a string then the string is split by `,` and any whitespace is trimmed
    - `path`: A string where values likes `%TEMP%` are expanded based on environment values. If the input value starts with `\\?\` then no expansion is run

    - `raw`: No conversions occur on the value passed in by Ansible
    - `sid`: Will convert Windows security identifier values or Windows account names to a SecurityIdentifier value
    - `str`: The value is converted to a string

When `type=dict`, or `type=list` and `elements=dict`, the following keys can also be set for that module option:

- `apply_defaults`: The value is based on the `options` spec defaults for that key if `True` and null if `False`. Only valid when the module option is not defined by the user and `type=dict`.
- `mutually_exclusive`: Same as the root level `mutually_exclusive` but validated against the values in the sub dict
- `options`: Same as the root level `options` but contains the valid options for the sub option
- `required_if`: Same as the root level `required_if` but validated against the values in the sub dict
- `required_by`: Same as the root level `required_by` but validated against the values in the sub dict
- `required_together`: Same as the root level `required_together` but validated against the values in the sub dict
- `required_one_of`: Same as the root level `required_one_of` but validated against the values in the sub dict

A module type can also be a delegate function that converts the value to whatever is required by the module option. For example the following snippet shows how to create a custom type that creates a `UInt64` value:

```
$spec = @{
    uint64_type = @{ type = [Func[[Object], [UInt64]]]{ [System.UInt64]::Parse($args[0]) } }
}
$uint64_type = $module.Params.uint64_type
```

When in doubt, look at some of the other core modules and see how things have been implemented there.

Sometimes there are multiple ways that Windows offers to complete a task; this is the order to favor when writing modules:

- Native Powershell cmdlets like `Remove-Item -Path C:\temp -Recurse`
- .NET classes like `[System.IO.Path]::GetRandomFileName()`
- WMI objects through the `New-CimInstance` cmdlet
- COM objects through `New-Object -ComObject` cmdlet
- Calls to native executables like `Secedit.exe`

PowerShell modules support a small subset of the `#Requires` options built into PowerShell as well as some Ansible-specific requirements specified by `#AnsibleRequires`. These statements can be placed at any point in the script, but are most commonly near the top. They are used to make it easier to state the requirements of the module without writing any of the checks. Each `requires` statement must be on its own line, but there can be multiple requires statements in one script.

These are the checks that can be used within Ansible modules:

- `#Requires -Module Ansible.ModuleUtils.<module_util>`: Added in Ansible 2.4, specifies a module_util to load in for the module execution.
- `#Requires -Version x.y`: Added in Ansible 2.5, specifies the version of PowerShell that is required by the module. The module will fail if this requirement is not met.
- `#AnsibleRequires -PowerShell <module_util>`: Added in Ansible 2.8, like `#Requires -Module`, this specifies a module_util to load in for module execution.
- `#AnsibleRequires -CSharpUtil <module_util>`: Added in Ansible 2.8, specifies a C# module_util to load in for the module execution.
- `#AnsibleRequires -OSVersion x.y`: Added in Ansible 2.5, specifies the OS build version that is required by the module and will fail if this requirement is not met. The actual OS version is derived from `[Environment]::OSVersion.Version`.
- `#AnsibleRequires -Become`: Added in Ansible 2.5, forces the exec runner to run the module with `become`, which is primarily used to bypass WinRM restrictions. If `ansible_become_user` is not specified then the `SYSTEM` account is used instead.

The `#AnsibleRequires -PowerShell` and `#AnsibleRequires -CSharpUtil` support further features such as:

- Importing a util contained in a collection (added in Ansible 2.9)
- Importing a util by relative names (added in Ansible 2.10)
- Specifying the util is optional by adding *-Optional* to the import declaration (added in Ansible 2.12).

See the below examples for more details:

```
# Imports the PowerShell Ansible.ModuleUtils.Legacy provided by Ansible itself
#AnsibleRequires -PowerShell Ansible.ModuleUtils.Legacy

# Imports the PowerShell my_util in the my_namesapce.my_name collection
#AnsibleRequires -PowerShell ansible_collections.my_namespace.my_name.plugins.module_utils.my_util

# Imports the PowerShell my_util that exists in the same collection as the current module
#AnsibleRequires -PowerShell ..module_utils.my_util

# Imports the PowerShell Ansible.ModuleUtils.Optional provided by Ansible if it exists.
# If it does not exist then it will do nothing.
#AnsibleRequires -PowerShell Ansible.ModuleUtils.Optional -Optional

# Imports the C# Ansible.Process provided by Ansible itself
#AnsibleRequires -CSharpUtil Ansible.Process

# Imports the C# my_util in the my_namespace.my_name collection
#AnsibleRequires -CSharpUtil ansible_collections.my_namespace.my_name.plugins.module_utils.my_util
```

```
# Imports the C# my_util that exists in the same collection as the current module
#AnsibleRequires -CSharpUtil ..module_utils.my_util

# Imports the C# Ansible.Optional provided by Ansible if it exists.
# If it does not exist then it will do nothing.
#AnsibleRequires -CSharpUtil Ansible.Optional -Optional
```

For optional require statements, it is up to the module code to then verify whether the util has been imported before trying to use it. This can be done by checking if a function or type provided by the util exists or not.

While both `#Requires -Module` and `#AnsibleRequires -PowerShell` can be used to load a PowerShell module it is recommended to use `#AnsibleRequires`. This is because `#AnsibleRequires` supports collection module utils, imports by relative util names, and optional util imports.

C# module utils can reference other C# utils by adding the line `using Ansible.<module_util>;` to the top of the script with all the other using statements.

Windows module utilities

Like Python modules, PowerShell modules also provide a number of module utilities that provide helper functions within PowerShell. These module_utils can be imported by adding the following line to a PowerShell module:

```
#Requires -Module Ansible.ModuleUtils.Legacy
```

This will import the module_util at `./lib/ansible/module_utils/powershell/Ansible.ModuleUtils.Legacy.psm1` and enable calling all of its functions. As of Ansible 2.8, Windows module utils can also be written in C# and stored at `lib/ansible/module_utils/csharp`. These module_utils can be imported by adding the following line to a PowerShell module:

```
#AnsibleRequires -CSharpUtil Ansible.Basic
```

This will import the module_util at `./lib/ansible/module_utils/csharp/Ansible.Basic.cs` and automatically load the types in the executing process. C# module utils can reference each other and be loaded together by adding the following line to the using statements at the top of the util:

```
using Ansible.Become;
```

There are special comments that can be set in a C# file for controlling the compilation parameters. The following comments can be added to the script;

- `//AssemblyReference -Name <assembly dll> [-CLR [Core|Framework]]`: The assembly DLL to reference during compilation, the optional `-CLR` flag can also be used to state whether to reference when running under .NET Core, Framework, or both (if omitted)
- `//NoWarn -Name <error id> [-CLR [Core|Framework]]`: A compiler warning ID to ignore when compiling the code, the optional `-CLR` works the same as above. A list of warnings can be found at Compiler errors

As well as this, the following pre-processor symbols are defined;

- `CORECLR`: This symbol is present when PowerShell is running through .NET Core
- `WINDOWS`: This symbol is present when PowerShell is running on Windows
- `UNIX`: This symbol is present when PowerShell is running on Unix

A combination of these flags help to make a module util interoperable on both .NET Framework and .NET Core, here is an example of them in action:

```
#if CORECLR
using Newtonsoft.Json;
#else
using System.Web.Script.Serialization;
#endif

//AssemblyReference -Name Newtonsoft.Json.dll -CLR Core
//AssemblyReference -Name System.Web.Extensions.dll -CLR Framework

// Ignore error CS1702 for all .NET types
//NoWarn -Name CS1702

// Ignore error CS1956 only for .NET Framework
//NoWarn -Name CS1956 -CLR Framework
```

The following is a list of module_utils that are packaged with Ansible and a general description of what they do:

- ArgvParser: Utility used to convert a list of arguments to an escaped string compliant with the Windows argument parsing rules.
- CamelConversion: Utility used to convert camelCase strings/lists/dicts to snake_case.
- CommandUtil: Utility used to execute a Windows process and return the stdout/stderr and rc as separate objects.
- FileUtil: Utility that expands on the `Get-ChildItem` and `Test-Path` to work with special files like `C:\pagefile.sys`.
- Legacy: General definitions and helper utilities for Ansible module.
- LinkUtil: Utility to create, remove, and get information about symbolic links, junction points and hard inks.

- SID: Utilities used to convert a user or group to a Windows SID and vice versa.

For more details on any specific module utility and their requirements, please see the Ansible module utilities source code.

PowerShell module utilities can be stored outside of the standard Ansible distribution for use with custom modules. Custom module_utils are placed in a folder called `module_utils` located in the root folder of the playbook or role directory.

C# module utilities can also be stored outside of the standard Ansible distribution for use with custom modules. Like PowerShell utils, these are stored in a folder called `module_utils` and the filename must end in the extension `.cs`, start with `Ansible.` and be named after the namespace defined in the util.

The below example is a role structure that contains two PowerShell custom module_utils called `Ansible.ModuleUtils.ModuleUtil1`, `Ansible.ModuleUtils.ModuleUtil2`, and a C# util containing the namespace `Ansible.CustomUtil`:

```
meta/
  main.yml
defaults/
  main.yml
module_utils/
  Ansible.ModuleUtils.ModuleUtil1.psm1
  Ansible.ModuleUtils.ModuleUtil2.psm1
  Ansible.CustomUtil.cs
tasks/
  main.yml
```

Each PowerShell module_util must contain at least one function that has been exported with `Export-ModuleMember` at the end of the file. For example

```
Export-ModuleMember -Function Invoke-CustomUtil, Get-CustomInfo
```

Exposing shared module options

PowerShell module utils can easily expose common module options that a module can use when building its argument spec. This allows common features to be stored and maintained in one location and have those features used by multiple modules with minimal effort. Any new features or bugfixes added to one of these utils are then automatically used by the various modules that call that util.

An example of this would be to have a module util that handles authentication and communication against an API This util can be used by multiple modules to expose a common set of module options like the API endpoint, username, password, timeout, cert validation, and so on without having to add those options to each module spec.

The standard convention for a module util that has a shared argument spec would have

- A `Get-<namespace.name.util name>Spec` function that outputs the common spec for a module
    - It is highly recommended to make this function name be unique to the module to avoid any conflicts with other utils that can be loaded
    - The format of the output spec is a Hashtable in the same format as the `$spec` used for normal modules
- A function that takes in an `AnsibleModule` object called under the `-Module` parameter which it can use to get the shared options

Because these options can be shared across various module it is highly recommended to keep the module option names and aliases in the shared spec as specific as they can be. For example do not have a util option called `password`, rather you should prefix it with a unique name like `acme_password`.

> **Warning**
>
> Failure to have a unique option name or alias can prevent the util being used by module that also use those names or aliases for its own options.

The following is an example module util called `ServiceAuth.psm1` in a collection that implements a common way for modules to authentication with a service.

```
Invoke-MyServiceResource {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory=$true)]
        [ValidateScript({ $_.GetType().FullName -eq 'Ansible.Basic.AnsibleModule' })]
        $Module,

        [Parameter(Mandatory=$true)]
        [String]
        $ResourceId,

        [String]
        $State = 'present'
    )

    # Process the common module options known to the util
    $params = @{
        ServerUri = $Module.Params.my_service_url
```

```
    }
    if ($Module.Params.my_service_username) {
        $params.Credential = Get-MyServiceCredential
    }

    if ($State -eq 'absent') {
        Remove-MyService @params -ResourceId $ResourceId
    } else {
        New-MyService @params -ResourceId $ResourceId
    }
}

Get-MyNamespaceMyCollectionServiceAuthSpec {
    # Output the util spec
    @{
        options = @{
            my_service_url = @{ type = 'str'; required = $true }
            my_service_username = @{ type = 'str' }
            my_service_password = @{ type = 'str'; no_log = $true }
        }

        required_together = @(
            ,@('my_service_username', 'my_service_password')
        )
    }
}

$exportMembers = @{
    Function = 'Get-MyNamespaceMyCollectionServiceAuthSpec', 'Invoke-MyServiceResource'
}
Export-ModuleMember @exportMembers
```

For a module to take advantage of this common argument spec it can be set out like

```
#!powershell

# Include the module util ServiceAuth.psm1 from the my_namespace.my_collection collection
#AnsibleRequires -PowerShell ansible_collections.my_namespace.my_collection.plugins.module_utils.ServiceAu

# Create the module spec like normal
$spec = @{
    options = @{
        resource_id = @{ type = 'str'; required = $true }
        state = @{ type = 'str'; choices = 'absent', 'present' }
    }
}

# Create the module from the module spec but also include the util spec to merge into our own.
$module = [Ansible.Basic.AnsibleModule]::Create($args, $spec, @(Get-MyNamespaceMyCollectionServiceAuthSpec

# Call the ServiceAuth module util and pass in the module object so it can access the module options.
Invoke-MyServiceResource -Module $module -ResourceId $module.Params.resource_id -State $module.params.sta

$module.ExitJson()
```

> **Note**
>
> Options defined in the module spec will always have precedence over a util spec. Any list values under the same key in a util spec will be appended to the module spec for that same key. Dictionary values will add any keys that are missing from the module spec and merge any values that are lists or dictionaries. This is similar to how the doc fragment plugins work when extending module documentation.

To document these shared util options for a module, create a doc fragment plugin that documents the options implemented by the module util and extend the module docs for every module that implements the util to include that fragment in its docs.

Windows playbook module testing

You can test a module with an Ansible playbook. For example:

- Create a playbook in any directory `touch testmodule.yml`.
- Create an inventory file in the same directory `touch hosts`.
- Populate the inventory file with the variables required to connect to a Windows host(s).
- Add the following to the new playbook file:

```
---
- name: test out windows module
  hosts: windows
  tasks:
  - name: test out module
    win_module:
```

```
            name: test name
```

- Run the playbook `ansible-playbook -i hosts testmodule.yml`

This can be useful for seeing how Ansible runs with the new module end to end. Other possible ways to test the module are shown below.

Windows debugging

Debugging a module currently can only be done on a Windows host. This can be useful when developing a new module or implementing bug fixes. These are some steps that need to be followed to set this up:

- Copy the module script to the Windows server
- Copy the folders `./lib/ansible/module_utils/powershell` and `./lib/ansible/module_utils/csharp` to the same directory as the script above
- Add an extra `#` to the start of any `#Requires -Module` lines in the module code, this is only required for any lines starting with `#Requires -Module`
- Add the following to the start of the module script that was copied to the server:

```
# Set $ErrorActionPreference to what's set during Ansible execution
$ErrorActionPreference = "Stop"

# Set the first argument as the path to a JSON file that contains the module args
$args = @("$($pwd.Path)\args.json")

# Or instead of an args file, set $complex_args to the pre-processed module args
$complex_args = @{
     ansible check mode = $false
    _ansible_diff = $false
    path = "C:\temp"
    state = "present"
}

# Import any C# utils referenced with '#AnsibleRequires -CSharpUtil' or 'using Ansible.;
# The $_csharp_utils entries should be the context of the C# util files and not the path
Import-Module -Name "$($pwd.Path)\powershell\Ansible.ModuleUtils.AddType.psm1"
$_csharp_utils = @(
    [System.IO.File]::ReadAllText("$($pwd.Path)\csharp\Ansible.Basic.cs")
)
Add-CSharpType -References $_csharp_utils -IncludeDebugInfo

# Import any PowerShell modules referenced with '#Requires -Module`
Import-Module -Name "$($pwd.Path)\powershell\Ansible.ModuleUtils.Legacy.psm1"

# End of the setup code and start of the module code
#!powershell
```

You can add more args to `$complex_args` as required by the module or define the module options through a JSON file with the structure:

```
{
    "ANSIBLE MODULE ARGS": {
        "_ansible_check_mode": false,
        " ansible diff": false,
        "path": "C:\\temp",
        "state": "present"
    }
}
```

There are multiple IDEs that can be used to debug a Powershell script, two of the most popular ones are

- Powershell ISE
- Visual Studio Code

To be able to view the arguments as passed by Ansible to the module follow these steps.

- Prefix the Ansible command with :envvar:`ANSIBLE_KEEP_REMOTE_FILES=1<ANSIBLE_KEEP_REMOTE_FILES>` to specify that Ansible should keep the exec files on the server.

  > **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\(ansible-devel)(docs)(docsite)(rst)(dev_guide)developing_modules_general_windows.rst`, **line 649**); *backlink*
  >
  > Unknown interpreted text role "envvar".

- Log onto the Windows server using the same user account that Ansible used to execute the module.
- Navigate to `%TEMP%\...` It should contain a folder starting with `ansible-tmp-`.
- Inside this folder, open the PowerShell script for the module.
- In this script is a raw JSON script under `$json_raw` which contains the module arguments under `module_args`. These args

can be assigned manually to the `$complex_args` variable that is defined on your debug script or put in the `args.json` file.

Windows unit testing

Currently there is no mechanism to run unit tests for Powershell modules under Ansible CI.

Windows integration testing

Integration tests for Ansible modules are typically written as Ansible roles. These test roles are located in `./test/integration/targets`. You must first set up your testing environment, and configure a test inventory for Ansible to connect to.

In this example we will set up a test inventory to connect to two hosts and run the integration tests for win_stat:

- Run the command `source ./hacking/env-setup` to prepare environment.
- Create a copy of `./test/integration/inventory.winrm.template` and name it `inventory.winrm`.
- Fill in entries under `[windows]` and set the required variables that are needed to connect to the host.
- :ref:`Install the required Python modules <windows_winrm>` to support WinRM and a configured authentication method.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\(ansible-devel)(docs)(docsite)(rst)(dev_guide)developing_modules_general_windows.rst, line 675); *backlink***
>
> Unknown interpreted text role "ref".

- To execute the integration tests, run `ansible-test windows-integration win_stat`; you can replace `win_stat` with the role you want to test.

This will execute all the tests currently defined for that role. You can set the verbosity level using the `-v` argument just as you would with ansible-playbook.

When developing tests for a new module, it is recommended to test a scenario once in check mode and twice not in check mode. This ensures that check mode does not make any changes but reports a change, as well as that the second run is idempotent and does not report changes. For example:

```
- name: remove a file (check mode)
  win_file:
    path: C:\temp
    state: absent
  register: remove_file_check
  check_mode: yes

- name: get result of remove a file (check mode)
  win_command: powershell.exe "if (Test-Path -Path 'C:\temp') { 'true' } else { 'false' }"
  register: remove_file_actual_check

- name: assert remove a file (check mode)
  assert:
    that:
    - remove file check is changed
    - remove_file_actual_check.stdout == 'true\r\n'

- name: remove a file
  win_file:
    path: C:\temp
    state: absent
  register: remove_file

- name: get result of remove a file
  win_command: powershell.exe "if (Test-Path -Path 'C:\temp') { 'true' } else { 'false' }"
  register: remove_file_actual

- name: assert remove a file
  assert:
    that:
    - remove_file is changed
    - remove_file_actual.stdout == 'false\r\n'

- name: remove a file (idempotent)
  win_file:
    path: C:\temp
    state: absent
  register: remove_file_again

- name: assert remove a file (idempotent)
  assert:
    that:
    - not remove_file_again is changed
```

Windows communication and development support

Join the `#ansible-devel` or `#ansible-windows` chat channels (using Matrix at ansible.im or using IRC at irc.libera.chat) for discussions about Ansible development for Windows.

For questions and discussions pertaining to using the Ansible product, use the `#ansible` channel.