Developing plugins

Plugins augment Ansible's core functionality with logic and features that are accessible to all modules. Ansible collections include a number of handy plugins, and you can easily write your own. All plugins must:

- be written in Python
- raise errors
- return strings in unicode
- conform to Ansible's configuration and documentation standards

Once you've reviewed these general guidelines, you can skip to the particular type of plugin you want to develop.

Writing plugins in Python

You must write your plugin in Python so it can be loaded by the `PluginLoader` and returned as a Python object that any module can use. Since your plugin will execute on the controller, you must write it in a :ref:`compatible version of Python <control_node_requirements>`.

> **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_plugins.rst`, line 23); *backlink***
>
> Unknown interpreted text role "ref".

Raising errors

You should return errors encountered during plugin execution by raising `AnsibleError()` or a similar class with a message describing the error. When wrapping other exceptions into error messages, you should always use the `to_native` Ansible function to ensure proper string compatibility across Python versions:

```
from ansible.module_utils.common.text.converters import to_native

try:
    cause_an_exception()
except Exception as e:
    raise AnsibleError('Something happened, this was original exception: %s' % to_native(e))
```

Since Ansible evaluates variables only when they are needed, filter and test plugins should propagate the exceptions `jinja2.exceptions.UndefinedError` and `AnsibleUndefinedVariable` to ensure undefined variables are only fatal when necessary.

Check the different AnsibleError objects and see which one applies best to your situation. Check the section on the specific plugin type you're developing for type-specific error handling details.

String encoding

You must convert any strings returned by your plugin into Python's unicode type. Converting to unicode ensures that these strings can run through Jinja2. To convert strings:

```
from ansible.module_utils.common.text.converters import to_text
result_string = to_text(result_string)
```

Plugin configuration & documentation standards

To define configurable options for your plugin, describe them in the `DOCUMENTATION` section of the python file. Callback and connection plugins have declared configuration requirements this way since Ansible version 2.4; most plugin types now do the same. This approach ensures that the documentation of your plugin's options will always be correct and up-to-date. To add a configurable option to your plugin, define it in this format:

```
options:
  option_name:
    description: describe this config option
    default: default value for this config option
    env:
      - name: NAME_OF_ENV_VAR
    ini:
      - section: section_of_ansible.cfg_where_this_config_option_is_defined
        key: key_used_in_ansible.cfg
    vars:
      - name: name_of_ansible_var
      - name: name_of_second_var
        version_added: X.x
    required: True/False
    type: boolean/float/integer/list/none/path/pathlist/pathspec/string/tmppath
    version_added: X.x
```

To access the configuration settings in your plugin, use `self.get_option(<option_name>)`. For the plugin types (such as 'become', 'cache', 'callback', 'cliconf', 'connection', 'httpapi', 'inventory', 'lookup', 'netconf', 'shell', and 'vars') that support embedded documentation, the controller pre-populates the settings. If you need to populate settings explicitly, use a `self.set_options()` call.

Configuration sources follow the precedence rules for values in Ansible. When there are multiple values from the same category, the value defined last takes precedence. For example, in the above configuration block, if both `name_of_ansible_var` and `name_of_second_var` are defined, the value of the `option_name` option will be the value of `name_of_second_var`. Refer to :ref:`general_precedence_rules` for further information.

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_plugins.rst, line 80); *backlink*

Unknown interpreted text role "ref".

Plugins that support embedded documentation (see :ref:`ansible-doc` for the list) should include well-formed doc strings. If you inherit from a plugin, you must document the options it takes, either via a documentation fragment or as a copy. See :ref:`module_documenting` for more information on correct documentation. Thorough documentation is a good idea even if you're developing a plugin for local use.

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_plugins.rst, line 82); *backlink*

Unknown interpreted text role "ref".

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_plugins.rst, line 82); *backlink*

Unknown interpreted text role "ref".

Developing particular plugin types

Action plugins

Action plugins let you integrate local processing and local data with module functionality.

To create an action plugin, create a new class with the Base(ActionBase) class as the parent:

```
from ansible.plugins.action import ActionBase

class ActionModule(ActionBase):
    pass
```

From there, execute the module using the `_execute_module` method to call the original module. After successful execution of the module, you can modify the module return data.

```
module_return = self._execute_module(module_name='<NAME_OF_MODULE>',
                                     module_args=module_args,
                                     task_vars=task_vars, tmp=tmp)
```

For example, if you wanted to check the time difference between your Ansible controller and your target machine(s), you could write an action plugin to check the local time and compare it to the return data from Ansible's `setup` module:

```
#!/usr/bin/python
# Make coding more python3-ish, this is required for contributions to Ansible
from __future__ import (absolute_import, division, print_function)
__metaclass__ = type

from ansible.plugins.action import ActionBase
from datetime import datetime


class ActionModule(ActionBase):
    def run(self, tmp=None, task_vars=None):
        super(ActionModule, self).run(tmp, task_vars)
        module_args = self._task.args.copy()
        module_return = self._execute_module(module_name='setup',
                                             module_args=module_args,
                                             task_vars=task_vars, tmp=tmp)
        ret = dict()
        remote_date = None
        if not module_return.get('failed'):
            for key, value in module_return['ansible_facts'].items():
                if key == 'ansible_date_time':
                    remote_date = value['iso8601']

        if remote_date:
            remote_date_obj = datetime.strptime(remote_date, '%Y-%m-%dT%H:%M:%SZ')
            time_delta = datetime.utcnow() - remote_date_obj
            ret['delta_seconds'] = time_delta.seconds
            ret['delta_days'] = time_delta.days
            ret['delta_microseconds'] = time_delta.microseconds

        return dict(ansible_facts=dict(ret))
```

This code checks the time on the controller, captures the date and time for the remote machine using the `setup` module, and calculates the difference between the captured time and the local time, returning the time delta in days, seconds and microseconds.

For practical examples of action plugins, see the source code for the action plugins included with Ansible Core

Cache plugins

Cache plugins store gathered facts and data retrieved by inventory plugins.

Import cache plugins using the cache_loader so you can use `self.set_options()` and `self.get_option(<option_name>)`. If you import a cache plugin directly in the code base, you can only access options via `ansible.constants`, and you break the cache plugin's ability to be used by an inventory plugin.

```
from ansible.plugins.loader import cache_loader
[...]
plugin = cache_loader.get('custom_cache', **cache_kwargs)
```

There are two base classes for cache plugins, `BaseCacheModule` for database-backed caches, and `BaseCacheFileModule` for file-backed caches.

To create a cache plugin, start by creating a new `CacheModule` class with the appropriate base class. If you're creating a plugin using an `__init__` method you should initialize the base class with any provided args and kwargs to be compatible with inventory plugin cache options. The base class calls `self.set_options(direct=kwargs)`. After the base class `__init__` method is called `self.get_option(<option_name>)` should be used to access cache options.

New cache plugins should take the options `_uri`, `_prefix`, and `_timeout` to be consistent with existing cache plugins.

```
from ansible.plugins.cache import BaseCacheModule
```

```
class CacheModule(BaseCacheModule):
    def __init__(self, *args, **kwargs):
        super(CacheModule, self).__init__(*args, **kwargs)
        self._connection = self.get_option('_uri')
        self._prefix = self.get_option('_prefix')
        self._timeout = self.get_option('_timeout')
```

If you use the `BaseCacheModule`, you must implement the methods `get`, `contains`, `keys`, `set`, `delete`, `flush`, and `copy`. The `contains` method should return a boolean that indicates if the key exists and has not expired. Unlike file-based caches, the `get` method does not raise a KeyError if the cache has expired.

If you use the `BaseFileCacheModule`, you must implement `_load` and `_dump` methods that will be called from the base class methods `get` and `set`.

If your cache plugin stores JSON, use `AnsibleJSONEncoder` in the `_dump` or `set` method and `AnsibleJSONDecoder` in the `_load` or `get` method.

For example cache plugins, see the source code for the cache plugins included with Ansible Core.

Callback plugins

Callback plugins add new behaviors to Ansible when responding to events. By default, callback plugins control most of the output you see when running the command line programs.

To create a callback plugin, create a new class with the Base(Callbacks) class as the parent:

```
from ansible.plugins.callback import CallbackBase

class CallbackModule(CallbackBase):
    pass
```

From there, override the specific methods from the CallbackBase that you want to provide a callback for. For plugins intended for use with Ansible version 2.0 and later, you should only override methods that start with `v2`. For a complete list of methods that you can override, please see `__init__.py` in the lib/ansible/plugins/callback directory.

The following is a modified example of how Ansible's timer plugin is implemented, but with an extra option so you can see how configuration works in Ansible version 2.4 and later:

```
# Make coding more python3-ish, this is required for contributions to Ansible
from __future__ import (absolute_import, division, print_function)
__metaclass__ = type

# not only visible to ansible-doc, it also 'declares' the options the plugin requires and how to configure them.
DOCUMENTATION = '''
callback: timer
callback_type: aggregate
requirements:
    - enable in configuration
short_description: Adds time to play stats
version_added: "2.0"  # for collections, use the collection version, not the Ansible version
description:
    - This callback just adds total play duration to the play stats.
options:
  format_string:
    description: format of the string shown to user at play end
    ini:
      - section: callback_timer
        key: format_string
    env:
      - name: ANSIBLE_CALLBACK_TIMER_FORMAT
    default: "Playbook run took %s days, %s hours, %s minutes, %s seconds"
'''
from datetime import datetime

from ansible.plugins.callback import CallbackBase


class CallbackModule(CallbackBase):
    """
    This callback module tells you how long your plays ran for.
    """
    CALLBACK_VERSION = 2.0
    CALLBACK_TYPE = 'aggregate'
    CALLBACK_NAME = 'namespace.collection_name.timer'

    # only needed if you ship it and don't want to enable by default
    CALLBACK_NEEDS_ENABLED = True

    def __init__(self):

      # make sure the expected objects are present, calling the base's __init__
      super(CallbackModule, self).__init__()

      # start the timer when the plugin is loaded, the first play should start a few milliseconds after.
      self.start_time = datetime.now()

    def _days_hours_minutes_seconds(self, runtime):
      ''' internal helper method for this callback '''
      minutes = (runtime.seconds // 60) % 60
      r_seconds = runtime.seconds - (minutes * 60)
      return runtime.days, runtime.seconds // 3600, minutes, r_seconds

    # this is only event we care about for display, when the play shows its summary stats; the rest are ignored by the base cl
    def v2_playbook_on_stats(self, stats):
      end_time = datetime.now()
      runtime = end_time - self.start_time

      # Shows the usage of a config option declared in the DOCUMENTATION variable. Ansible will have set it when it loads the
      # Also note the use of the display object to print to screen. This is available to all callbacks, and you should use thi
      self._display.display(self._plugin_options['format_string'] % (self._days_hours_minutes_seconds(runtime)))
```

Note that the `CALLBACK_VERSION` and `CALLBACK_NAME` definitions are required for properly functioning plugins for Ansible version 2.0 and later. `CALLBACK_TYPE` is mostly needed to distinguish 'stdout' plugins from the rest, since you can only load one plugin that writes to stdout.

For example callback plugins, see the source code for the callback plugins included with Ansible Core

New in ansible-core 2.11, callback plugins are notified (via `v2_playbook_on_task_start`) of :ref:`meta<meta_module>` tasks. By default, only explicit `meta` tasks that users list in their plays are sent to callbacks.

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_plugins.rst`, line 290); *backlink*

Unknown interpreted text role "ref".

There are also some tasks which are generated internally and implicitly at various points in execution. Callback plugins can opt-in to receiving these implicit tasks as well, by setting `self.wants_implicit_tasks = True`. Any `Task` object received by a callback hook will have an `.implicit` attribute, which can be consulted to determine whether the `Task` originated from within Ansible, or explicitly by the user.

Connection plugins

Connection plugins allow Ansible to connect to the target hosts so it can execute tasks on them. Ansible ships with many connection plugins, but only one can be used per host at a time. The most commonly used connection plugins are the `paramiko` SSH, native ssh (just called `ssh`), and `local` connection types. All of these can be used in playbooks and with `/usr/bin/ansible` to connect to remote machines.

Ansible version 2.1 introduced the `smart` connection plugin. The `smart` connection type allows Ansible to automatically select either the `paramiko` or `openssh` connection plugin based on system capabilities, or the `ssh` connection plugin if OpenSSH supports ControlPersist.

To create a new connection plugin (for example, to support SNMP, Message bus, or other transports), copy the format of one of the existing connection plugins and drop it into `connection` directory on your :ref:`local plugin path <local_plugins>`.

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_plugins.rst, line 303); *backlink*

Unknown interpreted text role "ref".

Connection plugins can support common options (such as the `--timeout` flag) by defining an entry in the documentation for the attribute name (in this case `timeout`). If the common option has a non-null default, the plugin should define the same default since a different default would be ignored.

For example connection plugins, see the source code for the connection plugins included with Ansible Core.

Filter plugins

Filter plugins manipulate data. They are a feature of Jinja2 and are also available in Jinja2 templates used by the `template` module. As with all plugins, they can be easily extended, but instead of having a file for each one you can have several per file. Most of the filter plugins shipped with Ansible reside in a `core.py`.

Filter plugins do not use the standard configuration and documentation system described above.

Since Ansible evaluates variables only when they are needed, filter plugins should propagate the exceptions `jinja2.exceptions.UndefinedError` and `AnsibleUndefinedVariable` to ensure undefined variables are only fatal when necessary.

```
try:
    cause_an_exception(with_undefined_variable)
except jinja2.exceptions.UndefinedError as e:
    raise AnsibleUndefinedVariable("Something happened, this was the original exception: %s" % to_native(e))
except Exception as e:
    raise AnsibleFilterError("Something happened, this was the original exception: %s" % to_native(e))
```

For example filter plugins, see the source code for the filter plugins included with Ansible Core.

Inventory plugins

Inventory plugins parse inventory sources and form an in-memory representation of the inventory. Inventory plugins were added in Ansible version 2.4.

You can see the details for inventory plugins in the :ref:`developing_inventory` page.

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_plugins.rst, line 338); *backlink*

Unknown interpreted text role "ref".

Lookup plugins

Lookup plugins pull in data from external data stores. Lookup plugins can be used within playbooks both for looping --- playbook language constructs like `with_fileglob` and `with_items` are implemented via lookup plugins --- and to return values into a variable or parameter.

Lookup plugins are very flexible, allowing you to retrieve and return any type of data. When writing lookup plugins, always return data of a consistent type that can be easily consumed in a playbook. Avoid parameters that change the returned data type. If there is a need to return a single value sometimes and a complex dictionary other times, write two different lookup plugins.

Ansible includes many :ref:`filters <playbooks_filters>` which can be used to manipulate the data returned by a lookup plugin. Sometimes it makes sense to do the filtering inside the lookup plugin, other times it is better to return results that can be filtered in the playbook. Keep in mind how the data will be referenced when determining the appropriate level of filtering to be done inside the lookup plugin.

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_plugins.rst, line 349); *backlink*

Unknown interpreted text role "ref".

Here's a simple lookup plugin implementation --- this lookup returns the contents of a text file as a variable:

```
# python 3 headers, required if submitting to Ansible
from __future__ import (absolute_import, division, print_function)
__metaclass__ = type

DOCUMENTATION = """
  lookup: file
  author: Daniel Hokka Zakrisson <daniel@hozac.com>
  version_added: "0.9"  # for collections, use the collection version, not the Ansible version
  short_description: read file contents
  description:
      - This lookup returns the contents from a file on the Ansible controller's file system.
  options:
    _terms:
      description: path(s) of files to read
```

```
      required: True
    option1:
      description:
            - Sample option that could modify plugin behaviour.
            - This one can be set directly ``option1='x'`` or in ansible.cfg, but can also use vars or environment.
      type: string
      ini:
        - section: file_lookup
          key: option1
  notes:
    - if read in variable context, the file can be interpreted as YAML if the content is valid to the parser.
    - this lookup does not understand globbing --- use the fileglob lookup instead.
"""
from ansible.errors import AnsibleError, AnsibleParserError
from ansible.plugins.lookup import LookupBase
from ansible.utils.display import Display

display = Display()

class LookupModule(LookupBase):

    def run(self, terms, variables=None, **kwargs):

      # First of all populate options,
      # this will already take into account env vars and ini config
      self.set_options(var_options=variables, direct=kwargs)

      # lookups in general are expected to both take a list as input and output a list
      # this is done so they work with the looping construct 'with_'.
      ret = []
      for term in terms:
          display.debug("File lookup term: %s" % term)

          # Find the file in the expected search path, using a class method
          # that implements the 'expected' search path for Ansible plugins.
          lookupfile = self.find_file_in_search_path(variables, 'files', term)

          # Don't use print or your own logging, the display class
          # takes care of it in a unified way.
          display.vvvv(u"File lookup using %s as file" % lookupfile)
          try:
              if lookupfile:
                  contents, show_data = self._loader._get_file_contents(lookupfile)
                  ret.append(contents.rstrip())
              else:
                  # Always use ansible error classes to throw 'final' exceptions,
                  # so the Ansible engine will know how to deal with them.
                  # The Parser error indicates invalid options passed
                  raise AnsibleParserError()
          except AnsibleParserError:
              raise AnsibleError("could not locate file in lookup: %s" % term)

          # consume an option: if this did something useful, you can retrieve the option value here
          if self.get_option('option1') == 'do something':
            pass

      return ret
```

The following is an example of how this lookup is called:

```
---
- hosts: all
  vars:
     contents: "{{ lookup('namespace.collection_name.file', '/etc/foo.txt') }}"
     contents_with_option: "{{ lookup('namespace.collection_name.file', '/etc/foo.txt', option1='donothing') }}"
  tasks:

     - debug:
         msg: the value of foo.txt is {{ contents }} as seen today {{ lookup('pipe', 'date +"%Y-%m-%d"') }}
```

For example lookup plugins, see the source code for the lookup plugins included with Ansible Core.

For more usage examples of lookup plugins, see :ref:`Using Lookups<playbooks_lookups>`.

> **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_plugins.rst, line 445); *backlink*
>
> Unknown interpreted text role "ref".

Test plugins

Test plugins verify data. They are a feature of Jinja2 and are also available in Jinja2 templates used by the `template` module. As with all plugins, they can be easily extended, but instead of having a file for each one you can have several per file. Most of the test plugins shipped with Ansible reside in a `core.py`. These are specially useful in conjunction with some filter plugins like `map` and `select`; they are also available for conditional directives like `when:`.

Test plugins do not use the standard configuration and documentation system described above.

Since Ansible evaluates variables only when they are needed, test plugins should propagate the exceptions `jinja2.exceptions.UndefinedError` and `AnsibleUndefinedVariable` to ensure undefined variables are only fatal when necessary.

```
try:
    cause_an_exception(with_undefined_variable)
except jinja2.exceptions.UndefinedError as e:
    raise AnsibleUndefinedVariable("Something happened, this was the original exception: %s" % to_native(e))
except Exception as e:
    raise AnsibleFilterError("Something happened, this was the original exception: %s" % to_native(e))
```

For example test plugins, see the source code for the test plugins included with Ansible Core.

Vars plugins

Vars plugins inject additional variable data into Ansible runs that did not come from an inventory source, playbook, or command line. Playbook constructs like 'host_vars' and 'group_vars' work using vars plugins.

Vars plugins were partially implemented in Ansible 2.0 and rewritten to be fully implemented starting with Ansible 2.4. Vars plugins are supported by collections starting with Ansible 2.10.

Older plugins used a `run` method as their main body/work:

```
def run(self, name, vault_password=None):
```

```
    pass # your code goes here
```

Ansible 2.0 did not pass passwords to older plugins, so vaults were unavailable. Most of the work now happens in the `get_vars` method which is called from the VariableManager when needed.

```
def get_vars(self, loader, path, entities):
    pass # your code goes here
```

The parameters are:

- loader: Ansible's DataLoader. The DataLoader can read files, auto-load JSON/YAML and decrypt vaulted data, and cache read files.
- path: this is 'directory data' for every inventory source and the current play's playbook directory, so they can search for data in reference to them. `get_vars` will be called at least once per available path.
- entities: these are host or group names that are pertinent to the variables needed. The plugin will get called once for hosts and again for groups.

This `get_vars` method just needs to return a dictionary structure with the variables.

Since Ansible version 2.4, vars plugins only execute as needed when preparing to execute a task. This avoids the costly 'always execute' behavior that occurred during inventory construction in older versions of Ansible. Since Ansible version 2.10, vars plugin execution can be toggled by the user to run when preparing to execute a task or after importing an inventory source.

You can create vars plugins that are not enabled by default using the class variable `REQUIRES_ENABLED`. If your vars plugin resides in a collection, it cannot be enabled by default. You must use `REQUIRES_ENABLED` in all collections-based vars plugins. To require users to enable your plugin, set the class variable `REQUIRES_ENABLED`:

```
class VarsModule(BaseVarsPlugin):
    REQUIRES_ENABLED = True
```

Include the `vars_plugin_staging` documentation fragment to allow users to determine when vars plugins run.

```
DOCUMENTATION = '''
    name: custom_hostvars
    version_added: "2.10"  # for collections, use the collection version, not the Ansible version
    short_description: Load custom host vars
    description: Load custom host vars
    options:
      stage:
        ini:
          - key: stage
            section: vars_custom_hostvars
        env:
          - name: ANSIBLE_VARS_PLUGIN_STAGE
    extends_documentation_fragment:
      - vars_plugin_staging
'''
```

For example vars plugins, see the source code for the vars plugins included with Ansible Core.

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_plugins.rst, line 534)

Unknown directive type "seealso".

```
.. seealso::

   :ref:`list_of_collections`
       Browse existing collections, modules, and plugins
   :ref:`developing_api`
       Learn about the Python API for task execution
   :ref:`developing_inventory`
       Learn about how to develop dynamic inventory sources
   :ref:`developing_modules_general`
       Learn about how to write Ansible modules
   `Mailing List <https://groups.google.com/group/ansible-devel>`_
       The development mailing list
   :ref:`communication_irc`
       How to join Ansible chat channels
```