Developing dynamic inventory

Ansible can pull inventory information from dynamic sources, including cloud sources, by using the supplied :ref:`inventory plugins <inventory_plugins>`. For details about how to pull inventory information, see :ref:`dynamic_inventory`. If the source you want is not currently covered by existing plugins, you can create your own inventory plugin as with any other plugin type.

> **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_inventory.rst`, **line 7**); ***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_inventory.rst`, **line 7**); ***backlink***
>
> Unknown interpreted text role "ref".

In previous versions, you had to create a script or program that could output JSON in the correct format when invoked with the proper arguments. You can still use and write inventory scripts, as we ensured backwards compatibility via the :ref:`script inventory plugin <script_inventory>` and there is no restriction on the programming language used. If you choose to write a script, however, you will need to implement some features yourself such as caching, configuration management, dynamic variable and group composition, and so on. If you use :ref:`inventory plugins <inventory_plugins>` instead, you can use the Ansible codebase and add these common features automatically.

> **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_inventory.rst`, **line 9**); ***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_inventory.rst`, **line 9**); ***backlink***
>
> Unknown interpreted text role "ref".

Topics

Inventory sources

Inventory sources are the input strings that inventory plugins work with. An inventory source can be a path to a file or to a script, or it can be raw data that the plugin can interpret.

The table below shows some examples of inventory plugins and the source types that you can pass to them with `-i` on the command line.

Plugin	Source
:ref:`host list <host_list_inventory>` **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_inventory.rst`, **line 33**); ***backlink*** Unknown interpreted text role "ref".	A comma-separated list of hosts

:ref:`yaml <yaml_inventory>` 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_inventory.rst, line 35); *backlink* Unknown interpreted text role "ref".	Path to a YAML format data file
:ref:`constructed <constructed_inventory>` 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_inventory.rst, line 37); *backlink* Unknown interpreted text role "ref".	Path to a YAML configuration file
:ref:`ini <ini_inventory>` 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_inventory.rst, line 39); *backlink* Unknown interpreted text role "ref".	Path to an INI formatted data file
:ref:`virtualbox <virtualbox_inventory>` 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_inventory.rst, line 41); *backlink* Unknown interpreted text role "ref".	Path to a YAML configuration file
:ref:`script plugin <script_inventory>` 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_inventory.rst, line 43); *backlink* Unknown interpreted text role "ref".	Path to an executable that outputs JSON

Inventory plugins

Like most plugin types (except modules), inventory plugins must be developed in Python. They execute on the controller and should therefore adhere to the :ref:`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_inventory.rst, line 51); *backlink*

Unknown interpreted text role "ref".

Most of the documentation in :ref:`developing_plugins` also applies here. You should read that document first for a general understanding and then come back to this document for specifics on inventory 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_inventory.rst, line 53); *backlink*

Unknown interpreted text role "ref".

Normally, inventory plugins are executed at the start of a run, and before the playbooks, plays, or roles are loaded. However, you can use the `meta: refresh_inventory` task to clear the current inventory and execute the inventory plugins again, and this task will generate a new inventory.

If you use the persistent cache, inventory plugins can also use the configured cache plugin to store and retrieve data. Caching inventory avoids making repeated and costly external calls.

Developing an inventory plugin

The first thing you want to do is use the base class:

```
from ansible.plugins.inventory import BaseInventoryPlugin

class InventoryModule(BaseInventoryPlugin):

    NAME = 'myplugin'  # used internally by Ansible, it should match the file name but not required
```

If the inventory plugin is in a collection, the NAME should be in the 'namespace.collection_name.myplugin' format. The base class has a couple of methods that each plugin should implement and a few helpers for parsing the inventory source and updating the inventory.

After you have the basic plugin working, you can incorporate other features by adding more base classes:

```
from ansible.plugins.inventory import BaseInventoryPlugin, Constructable, Cacheable

class InventoryModule(BaseInventoryPlugin, Constructable, Cacheable):

    NAME = 'myplugin'
```

For the bulk of the work in a plugin, we mostly want to deal with 2 methods `verify_file` and `parse`.

verify_file method

Ansible uses this method to quickly determine if the inventory source is usable by the plugin. The determination does not need to be 100% accurate, as there might be an overlap in what plugins can handle and by default Ansible will try the enabled plugins as per their sequence.

```
def verify_file(self, path):
    ''' return true/false if this is possibly a valid file for this plugin to consume '''
    valid = False
    if super(InventoryModule, self).verify_file(path):
        # base class verifies that file exists and is readable by current user
        if path.endswith(('virtualbox.yaml', 'virtualbox.yml', 'vbox.yaml', 'vbox.yml')):
            valid = True
    return valid
```

In the above example, from the :ref:`virtualbox inventory plugin <virtualbox_inventory>`, we screen for specific file name patterns to avoid attempting to consume any valid YAML file. You can add any type of condition here, but the most common one is 'extension matching'. If you implement extension matching for YAML configuration files, the path suffix <plugin_name>.<yml|yaml> should be accepted. All valid extensions should be documented in the plugin description.

> **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_inventory.rst, line 107); *backlink*
>
> Unknown interpreted text role "ref".

The following is another example that does not use a 'file' but the inventory source string itself, from the :ref:`host list <host_list_inventory>` 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_inventory.rst, line 109); *backlink*
>
> Unknown interpreted text role "ref".

```
def verify_file(self, path):
    ''' don't call base class as we don't expect a path, but a host list '''
    host_list = path
    valid = False
    b_path = to_bytes(host_list, errors='surrogate_or_strict')
    if not os.path.exists(b_path) and ',' in host_list:
        # the path does NOT exist and there is a comma to indicate this is a 'host list'
        valid = True
    return valid
```

This method is just to expedite the inventory process and avoid unnecessary parsing of sources that are easy to filter out before causing a parse error.

parse method

This method does the bulk of the work in the plugin. It takes the following parameters:

- inventory: inventory object with existing data and the methods to add hosts/groups/variables to inventory
- loader: Ansible's DataLoader. The DataLoader can read files, auto load JSON/YAML and decrypt vaulted data, and cache read files.
- path: string with inventory source (this is usually a path, but is not required)
- cache: indicates whether the plugin should use or avoid caches (cache plugin and/or loader)

The base class does some minimal assignment for reuse in other methods.

```
def parse(self, inventory, loader, path, cache=True):

    self.loader = loader
    self.inventory = inventory
    self.templar = Templar(loader=loader)
```

It is up to the plugin now to parse the provided inventory source and translate it into Ansible inventory. To facilitate this, the example below uses a few helper functions:

```
NAME = 'myplugin'
```

```
def parse(self, inventory, loader, path, cache=True):

    # call base method to ensure properties are available for use with other helper methods
    super(InventoryModule, self).parse(inventory, loader, path, cache)

    # this method will parse 'common format' inventory sources and
    # update any options declared in DOCUMENTATION as needed
    config = self._read_config_data(path)

    # if NOT using _read_config_data you should call set_options directly,
    # to process any defined configuration for this plugin,
    # if you don't define any options you can skip
    #self.set_options()

    # example consuming options from inventory source
    mysession = apilib.session(user=self.get option('api user'),
                               password=self.get_option('api_pass'),
                               server=self.get_option('api_server')
    )


    # make requests to get data to feed into inventory
    mydata = mysession.getitall()

    #parse data and create inventory objects:
    for colo in mydata:
        for server in mydata[colo]['servers']:
            self.inventory.add_host(server['name'])
            self.inventory.set_variable(server['name'], 'ansible_host', server['external_ip'])
```

The specifics will vary depending on API and structure returned. Remember that if you get an inventory source error or any other issue, you should `raise AnsibleParserError` to let Ansible know that the source was invalid or the process failed.

For examples on how to implement an inventory plugin, see the source code here: lib/ansible/plugins/inventory.

inventory cache

To cache the inventory, extend the inventory plugin documentation with the inventory_cache documentation fragment and use the Cacheable base class.

```
extends documentation_fragment:
  - inventory_cache
```

```
class InventoryModule(BaseInventoryPlugin, Constructable, Cacheable):

    NAME = 'myplugin'
```

Next, load the cache plugin specified by the user to read from and update the cache. If your inventory plugin uses YAML-based configuration files and the `_read_config_data` method, the cache plugin is loaded within that method. If your inventory plugin does not use `_read_config_data`, you must load the cache explicitly with `load_cache_plugin`.

```
NAME = 'myplugin'

def parse(self, inventory, loader, path, cache=True):
    super(InventoryModule, self).parse(inventory, loader, path)

    self.load_cache_plugin()
```

Before using the cache plugin, you must retrieve a unique cache key by using the `get_cache_key` method. This task needs to be done by all inventory modules using the cache, so that you don't use/overwrite other parts of the cache.

```
def parse(self, inventory, loader, path, cache=True):
    super(InventoryModule, self).parse(inventory, loader, path)

    self.load_cache_plugin()
    cache_key = self.get_cache_key(path)
```

Now that you've enabled caching, loaded the correct plugin, and retrieved a unique cache key, you can set up the flow of data between the cache and your inventory using the `cache` parameter of the `parse` method. This value comes from the inventory manager and indicates whether the inventory is being refreshed (such as via `--flush-cache` or the meta task `refresh_inventory`). Although the cache shouldn't be used to populate the inventory when being refreshed, the cache should be updated with the new inventory if the user has enabled caching. You can use `self._cache` like a dictionary. The following pattern allows refreshing the inventory to work in conjunction with caching.

```
def parse(self, inventory, loader, path, cache=True):
    super(InventoryModule, self).parse(inventory, loader, path)

    self.load cache plugin()
    cache_key = self.get_cache_key(path)

    # cache may be True or False at this point to indicate if the inventory is being refreshed
    # get the user's cache option too to see if we should save the cache if it is changing
    user_cache_setting = self.get_option('cache')

    # read if the user has caching enabled and the cache isn't being refreshed
    attempt_to_read_cache = user_cache_setting and cache
    # update if the user has caching enabled and the cache is being refreshed; update this value to True if the cache h
    cache_needs_update = user_cache_setting and not cache

    # attempt to read the cache if inventory isn't being refreshed and the user has caching enabled
    if attempt_to_read_cache:
        try:
            results = self._cache[cache_key]
        except KeyError:
            # This occurs if the cache_key is not in the cache or if the cache_key expired, so the cache needs to be up
            cache_needs_update = True
    if not attempt_to_read_cache or cache_needs_update:
        # parse the provided inventory source
        results = self.get_inventory()
    if cache_needs_update:
        self._cache[cache_key] = results
```

```
        # submit the parsed data to the inventory object (add_host, set_variable, etc)
        self.populate(results)
```

After the `parse` method is complete, the contents of `self._cache` is used to set the cache plugin if the contents of the cache have changed.

You have three other cache methods available:

- `set_cache_plugin` forces the cache plugin to be set with the contents of `self._cache`, before the `parse` method completes
- `update_cache_if_changed` sets the cache plugin only if `self._cache` has been modified, before the `parse` method completes
- `clear_cache` flushes the cache, ultimately by calling the cache plugin's `flush()` method, whose implementation is dependent upon the particular cache plugin in use. Note that if the user is using the same cache backend for facts and inventory, both will get flushed. To avoid this, the user can specify a distinct cache backend in their inventory plugin configuration.

constructed features

Inventory plugins can create host variables and groups from Jinja2 expressions and variables by using features from the `constructed` inventory plugin. To do this, use the `Constructable` base class and extend the inventory plugin's documentation with the `constructed` documentation fragment.

```
extends_documentation_fragment:
  - constructed
```

```
class InventoryModule(BaseInventoryPlugin, Constructable):

    NAME = 'ns.coll.myplugin'
```

The three main options from the `constructed` documentation fragment are `compose`, `keyed_groups`, and `groups`. See the `constructed` inventory plugin for examples on using these. `compose` is a dictionary of variable names and Jinja2 expressions. Once a host is added to inventory and any initial variables have been set, call the method `_set_composite_vars` to add composed host variables. If this is done before adding `keyed_groups` and `groups`, the group generation will be able to use the composed variables.

```
def add_host(self, hostname, host_vars):
    self.inventory.add_host(hostname, group='all')

    for var_name, var_value in host_vars.items():
        self.inventory.set_variable(hostname, var_name, var_value)

    # Determines if composed variables or groups using nonexistent variables is an error
    strict = self.get_option('strict')

    # Add variables created by the user's Jinja2 expressions to the host
    self._set_composite_vars(self.get_option('compose'), host_vars, hostname, strict=True)

    # The following two methods combine the provided variables dictionary with the latest host variables
    # Using these methods after _set_composite_vars() allows groups to be created with the composed variables
    self._add_host_to_composed_groups(self.get_option('groups'), host_vars, hostname, strict=strict)
    self._add_host_to_keyed_groups(self.get_option('keyed_groups'), host_vars, hostname, strict=strict)
```

By default, group names created with `_add_host_to_composed_groups()` and `_add_host_to_keyed_groups()` are valid Python identifiers. Invalid characters are replaced with an underscore `_`. A plugin can change the sanitization used for the constructed features by setting `self._sanitize_group_name` to a new function. The core engine also does sanitization, so if the custom function is less strict it should be used in conjunction with the configuration setting `TRANSFORM_INVALID_GROUP_CHARS`.

```
from ansible.inventory.group import to_safe_group_name

class InventoryModule(BaseInventoryPlugin, Constructable):

    NAME = 'ns.coll.myplugin'

    @staticmethod
    def custom_sanitizer(name):
        return to_safe_group_name(name, replacer='')

    def parse(self, inventory, loader, path, cache=True):
        super(InventoryModule, self).parse(inventory, loader, path)

        self._sanitize_group_name = custom_sanitizer
```

Common format for inventory sources

To simplify development, most plugins use a standard YAML-based configuration file as the inventory source. The file has only one required field `plugin`, which should contain the name of the plugin that is expected to consume the file. Depending on other common features used, you might need other fields, and you can add custom options in each plugin as required. For example, if you use the integrated caching, `cache_plugin`, `cache_timeout` and other cache-related fields could be present.

The 'auto' plugin

From Ansible 2.5 onwards, we include the :ref:`auto inventory plugin <auto_inventory>` and enable it by default. If the `plugin` field in your standard configuration file matches the name of your inventory plugin, the `auto` inventory plugin will load your plugin. The 'auto' plugin makes it easier to use your plugin without having to update configurations.

> **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_inventory.rst`, line 343); *backlink*
>
> Unknown interpreted text role "ref".

Inventory scripts

Even though we now have inventory plugins, we still support inventory scripts, not only for backwards compatibility but also to allow users to use other programming languages.

Inventory script conventions

Inventory scripts must accept the `--list` and `--host <hostname>` arguments. Although other arguments are allowed, Ansible will not use them. Such arguments might still be useful for executing the scripts directly.

When the script is called with the single argument `--list`, the script must output to stdout a JSON object that contains all the groups to be managed. Each group's value should be either an object containing a list of each host, any child groups, and potential group variables, or simply a list of hosts:

```
{
    "group001": {
        "hosts": ["host001", "host002"],
        "vars": {
            "var1": true
        },
        "children": ["group002"]
    },
    "group002": {
        "hosts": ["host003","host004"],
        "vars": {
            "var2": 500
        },
        "children":[]
    }

}
```

If any of the elements of a group are empty, they may be omitted from the output.

When called with the argument `--host <hostname>` (where <hostname> is a host from above), the script must print a JSON object, either empty or containing variables to make them available to templates and playbooks. For example:

```
{
    "VAR001": "VALUE",
    "VAR002": "VALUE"
}
```

Printing variables is optional. If the script does not print variables, it should print an empty JSON object.

Tuning the external inventory script

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_inventory.rst, line 404)

Unknown directive type "versionadded".

```
.. versionadded:: 1.3
```

The stock inventory script system mentioned above works for all versions of Ansible, but calling `--host` for every host can be rather inefficient, especially if it involves API calls to a remote subsystem.

To avoid this inefficiency, if the inventory script returns a top-level element called "_meta", it is possible to return all the host variables in a single script execution. When this meta element contains a value for "hostvars", the inventory script will not be invoked with `--host` for each host. This behavior results in a significant performance increase for large numbers of hosts.

The data to be added to the top-level JSON object looks like this:

```
{

    # results of inventory script as above go here
    # ...

    " meta": {
        "hostvars": {
            "host001": {
                "var001" : "value"
            },
            "host002": {
                "var002": "value"
            }
        }
    }
}
```

To satisfy the requirements of using `_meta`, to prevent ansible from calling your inventory with `--host` you must at least populate `_meta` with an empty `hostvars` object. For example:

```
{

    # results of inventory script as above go here
    # ...

    "_meta": {
        "hostvars": {}
    }
}
```

If you intend to replace an existing static inventory file with an inventory script, it must return a JSON object which contains an 'all' group that includes every host in the inventory as a member and every group in the inventory as a child. It should also include an 'ungrouped' group which contains all hosts which are not members of any other group. A skeleton example of this JSON object is:

```
{
    "_meta": {
      "hostvars": {}
    },
    "all": {
      "children": [
        "ungrouped"
      ]
```

```
    },
    "ungrouped": {
      "children": [
      ]
    }
}
```

An easy way to see how this should look is using :ref:`ansible-inventory`, which also supports `--list` and `--host` parameters like an inventory script would.

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_inventory.rst`, line 469); *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_inventory.rst`, line 471)

Unknown directive type "seealso".

```
.. seealso::

   :ref:`developing_api`
       Python API to Playbooks and Ad Hoc Task Execution
   :ref:`developing_modules_general`
       Get started with developing a module
   :ref:`developing_plugins`
       How to develop plugins
   `AWX <https://github.com/ansible/awx>`_
       REST API endpoint and GUI for Ansible, syncs with dynamic inventory
   `Development Mailing List <https://groups.google.com/group/ansible-devel>`_
       Mailing list for development topics
   :ref:`communication_irc`
       How to join Ansible chat channels
```