Developing network plugins

- Network connection plugins
- Developing httpapi plugins
  - Making requests
  - Authenticating
  - Error handling
- Developing NETCONF plugins
- Developing network_cli plugins
- Developing cli_parser plugins in a collection

You can extend the existing network modules with custom plugins in your collection.

Network connection plugins

Each network connection plugin has a set of its own plugins which provide a specification of the connection for a particular set of devices. The specific plugin used is selected at runtime based on the value of the `ansible_network_os` variable assigned to the host. This variable should be set to the same value as the name of the plugin to be loaded. Thus, `ansible_network_os=nxos` will try to load a plugin in a file named `nxos.py`, so it is important to name the plugin in a way that will be sensible to users.

Public methods of these plugins may be called from a module or module_utils with the connection proxy object just as other connection methods can. The following is a very simple example of using such a call in a module_utils file so it may be shared with other modules.

```
from ansible.module_utils.connection import Connection

def get_config(module):
    # module is your AnsibleModule instance.
    connection = Connection(module._socket_path)

    # You can now call any method (that doesn't start with '_') of the connection
    # plugin or its platform-specific plugin
    return connection.get_config()
```

Developing httpapi plugins

`httpapi plugins <httpapi_plugins>` serve as adapters for various HTTP(S) APIs for use with the `httpapi` connection plugin. They should implement a minimal set of convenience methods tailored to the API you are attempting to use.

Specifically, there are a few methods that the `httpapi` connection plugin expects to exist.

Making requests

The `httpapi` connection plugin has a `send()` method, but an httpapi plugin needs a `send_request(self, data, **message_kwargs)` method as a higher-level wrapper to `send()`. This method should prepare requests by adding fixed values like common headers or URL root paths. This method may do more complex work such as turning data into formatted payloads, or determining which path or method to request. It may then also unpack responses to be more easily consumed by the caller.

```
from ansible.module_utils.six.moves.urllib.error import HTTPError

def send_request(self, data, path, method='POST'):
    # Fixed headers for requests
    headers = {'Content-Type': 'application/json'}
    try:
        response, response_content = self.connection.send(path, data, method=method,
```

```
headers=headers)
    except HTTPError as exc:
        return exc.code, exc.read()

    # handle_response (defined separately) will take the format returned by the device
    # and transform it into something more suitable for use by modules.
    # This may be JSON text to Python dictionaries, for example.
    return handle_response(response_content)
```

Authenticating

By default, all requests will authenticate with HTTP Basic authentication. If a request can return some kind of token to stand in place of HTTP Basic, the `update_auth(self, response, response_text)` method should be implemented to inspect responses for such tokens. If the token is meant to be included with the headers of each request, it is sufficient to return a dictionary which will be merged with the computed headers for each request. The default implementation of this method does exactly this for cookies. If the token is used in another way, say in a query string, you should instead save that token to an instance variable, where the `send_request()` method (above) can add it to each request

```
def update_auth(self, response, response_text):
    cookie = response.info().get('Set-Cookie')
    if cookie:
        return {'Cookie': cookie}

    return None
```

If instead an explicit login endpoint needs to be requested to receive an authentication token, the `login(self, username, password)` method can be implemented to call that endpoint. If implemented, this method will be called once before requesting any other resources of the server. By default, it will also be attempted once when a HTTP 401 is returned from a request.

```
def login(self, username, password):
    login_path = '/my/login/path'
    data = {'user': username, 'password': password}

    response = self.send_request(data, path=login_path)
    try:
        # This is still sent as an HTTP header, so we can set our connection's _auth
        # variable manually. If the token is returned to the device in another way,
        # you will have to keep track of it another way and make sure that it is sent
```

```
        # with the rest of the request from send_request()
        self.connection._auth = {'X-api-token': response['token']}
    except KeyError:
        raise AnsibleAuthenticationFailure(message="Failed to acquire login token.")
```

Similarly, `logout(self)` can be implemented to call an endpoint to invalidate and/or release the current token, if such an endpoint exists. This will be automatically called when the connection is closed (and, by extension, when reset).

```
def logout(self):
    logout_path = '/my/logout/path'
    self.send_request(None, path=logout_path)

    # Clean up tokens
    self.connection._auth = None
```

Error handling

The `handle_httperror(self, exception)` method can deal with status codes returned by the server. The return value indicates how the plugin will continue with the request:

- A value of `true` means that the request can be retried. This may be used to indicate a transient error, or one that has been resolved. For example, the default implementation will try to call `login()` when presented with a 401, and return `true` if successful.
- A value of `false` means that the plugin is unable to recover from this response. The status code will be raised as an exception to the calling module.
- Any other value will be taken as a nonfatal response from the request. This may be useful if the server returns error messages in the body of the response. Returning the original exception is usually sufficient in this case, as HTTPError objects have the same interface as a successful response.

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

Developing NETCONF plugins

The `netconf <netconf_connection>` connection plugin provides a connection to remote devices over the `SSH NETCONF` subsystem. Network devices typically use this connection plugin to send and receive `RPC` calls over `NETCONF`.

The `netconf` connection plugin uses the `ncclient` Python library under the hood to initiate a NETCONF session with a NETCONF-enabled remote network device. `ncclient` also executes NETCONF RPC requests and receives responses. You must install the `ncclient` on the local Ansible control node.

To use the `netconf` connection plugin for network devices that support standard NETCONF (RFC 6241) operations such as `get`, `get-config`, `edit-config`, set `ansible_network_os=default`. You can use `netconf_get <netconf_get_module>`, `netconf_config <netconf_config_module>` and `netconf_rpc <netconf_rpc_module>` modules to talk to a NETCONF enabled remote host.

As a contributor and user, you should be able to use all the methods under the `NetconfBase` class if your device supports standard NETCONF. You can contribute a new plugin if the device you are working with has a vendor specific NETCONF RPC. To support a vendor specific NETCONF RPC, add the implementation in the network OS specific NETCONF plugin.

For Junos for example:

- See the vendor-specific Junos RPC methods implemented in `plugins/netconf/junos.py`.
- Set the value of `ansible_network_os` to the name of the netconf plugin file, that is `junos` in this case.

Developing network_cli plugins

The `network_cli <network_cli_connection>` connection type uses `paramiko_ssh` under the hood which creates a pseudo terminal to send commands and receive responses.

> [!NOTE] The Paramiko-based connection plugin is deprecated. Use the libssh connection plugin instead. For details on migrating from Paramiko to libssh, see Migrating to ssh_type libssh for ansible.netcommon.network_cli across Ansible networking collections.

`network_cli` loads two platform specific plugins based on the value of `ansible_network_os`:

- Terminal plugin (for example `plugins/terminal/ios.py`) - Controls the parameters related to terminal, such as setting terminal length and width, page disabling and privilege escalation. Also defines regex to identify the command prompt and error prompts.
- `cliconf_plugins` (for example, `ios cliconf <ios_cliconf>`) - Provides an abstraction layer for low level send and receive operations. For example, the `edit_config()` method ensures that the prompt is in `config` mode before executing configuration commands.

To contribute a new network operating system to work with the `network_cli` connection, implement the `cliconf` and `terminal` plugins for that network OS.

The plugins can reside in:

- Adjacent to playbook in folders

```
cliconf_plugins/
terminal_plugins/
```

- Roles

```
myrole/cliconf_plugins/
myrole/terminal_plugins/
```

- Collections

```
myorg/mycollection/plugins/terminal/
myorg/mycollection/plugins/cliconf/
```

The user can also set the `DEFAULT_CLICONF_PLUGIN_PATH` to configure the `cliconf` plugin path.

After adding the `cliconf` and `terminal` plugins in the expected locations, users can:

- Use the `cli_command <cli_command_module>` to run an arbitrary command on the network device.
- Use the `cli_config <cli_config_module>` to implement configuration changes on the remote hosts without platform-specific modules.

Developing cli_parser plugins in a collection

You can use `cli_parse` as an entry point for a cli_parser plugin in your own collection.

The following sample shows the start of a custom cli_parser plugin:

```
from ansible_collections.ansible.netcommon.plugins.module_utils.cli_parser.cli_parserbase
import (
    CliParserBase,
)
```

```
class CliParser(CliParserBase):
    """ Sample cli_parser plugin
    """

    # Use the following extension when loading a template
    DEFAULT_TEMPLATE_EXTENSION = "txt"
    # Provide the contents of the template to the parse function
    PROVIDE_TEMPLATE_CONTENTS = True

    def myparser(text, template_contents):
      # parse the text using the template contents
      return {...}

    def parse(self, *_args, **kwargs):
        """ Standard entry point for a cli_parse parse execution

        :return: Errors or parsed text as structured data
        :rtype: dict

        :example:

        The parse function of a parser should return a dict:
        {"errors": [a list of errors]}
        or
        {"parsed": obj}
        """
        template_contents = kwargs["template_contents"]
        text = self._task_args.get("text")
        try:
            parsed = myparser(text, template_contents)
        except Exception as exc:
            msg = "Custom parser returned an error while parsing. Error: {err}"
            return {"errors": [msg.format(err=to_native(exc))]}
        return {"parsed": parsed}
```

The following task uses this custom cli_parser plugin:

```
- name: Use a custom cli_parser
  ansible.netcommon.cli_parse:
    command: ls -l
```

```
    parser:
      name: my_organization.my_collection.custom_parser
```

To develop a custom plugin: - Each cli_parser plugin requires a `CliParser` class. - Each cli_parser plugin requires a `parse` function. - Always return a dictionary with `errors` or `parsed`. - Place the custom cli_parser in plugins/cli_parsers directory of the collection. - See the current cli_parsers for examples to follow.

- `cli_parsing`

□□□ #2
□ □□□ □□□ 2026-10-31 14:37:51 UTC
□ □□□ □□□ 2026-10-31 14:37:52 UTC