APRIL 06, 2015

Running Ansible Programmatically

The Use Case

I've recently had a need to run Ansible programmatically. In this case, I was spinning up EC2 servers for potential customers of what usually is an on-premise application. These servers are used for trials, but can easily be turned into a paid-for production instance.

There are a few steps in this process:

1. Create an EC2 server. This is done via the AWS API.
2. Create a delayed queue job to poll if the server is spin up and running
   - If the server is running, create a few related services, assign the server an elastic IP address, and then setup DNS for a subdomain. If all goes well, the last step is to create another job that kicks off Ansible.
3. Run Ansible to provision the server using some variables generated and some input by the customer, most of which cannot be known ahead of time.

Benefits

One way we may have been able to do this was to take the user input and generated data and write it to files. Ansible could then be run "normally", via a command-line call. However, we chose to instead fire Ansible programmatically within some Python code as it has a few benefits:

- **Dynamically control Inventory** - We don't know the EC2 server information ahead of time
- **Automation with unknown/changing variables** - Customer's choose some variables/data we won't know ahead of time
- **More control over error handling** - We can get more context when errors occur within the queue jobs
- **Extend Ansible behavior** - We can add Ansible plugins to save output to an aggregated log (Logstash, for example) or database for auditing

All of this seemed easier when using Ansible programmatically. Ansible has a clean and fairly simple API, making it fairly easy to follow and figure out.

Install Dependencies

We installed Ansible via pip into a virtual environment, but you can install Ansible in any fashion that suits your needs. We're building on Ubuntu servers, so in Ubuntu and likely Debian, we install it like so:

```
# Get pip
sudo apt-get install -y python-pip

# Get virtualenv and create one
sudo pip install virtualenv
cd /path/to/runner/script
virtualenv ./.env
source ./env/bin/activate

# Install Ansible into the virtualenv
pip install ansible
```

Then, with the environment active, we call upon Ansible from our Python scripts.

Ansible Config

First I create an `ansible.cfg` file. Ansible will find this file via an `ANSIBLE_CONFIG` environment variable. Using this is easier than setting the configurations in code, due to the order this configuration file is loaded in and how Ansible reads in constants (constants set by configuration).

```
[defaults]
log_path = /path/to/ansible.log
callback_plugins = /path/to/our/ansible/plugins/callback_plugins:~/.ansible/plugins/callback_plugins/:/usr/share/ansible
_plugins/callback_plugins

[ssh_connection]
ssh_args = -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no -o IdentitiesOnly=yes -o ControlMaster=auto -o Co
ntrolPersist=60s
control_path = /home/some_user_name/.ansible/cp/ansible-ssh-%%h-%%p-%%r
```

Here's what is set here:

- `log_path` - Uses the internal log plugin to log all actions to a file. I may turned this off in production in favor of our own log plugin which will send logs to another source (database or a log aggregator).
- `callback_plugins` - The file path a custom logging plugin (or any plugin!) would be autoloaded from, if I used any. I add my own path in first, and then append the default paths after.
- `ssh_args` - Sets SSH options as you would normally set with the `-o` flag. This controls how Ansible connects to the host server. The ones I used will prevent the prompt that asks if this is a trusted host (be cautious with doing that!) and ensure Ansible uses our private key to access the server. Check out the video on Logging in with SSH (https://serversforhackers.com/video/logging-in-with-ssh) to see some examples of those options used with the SSH command.
- `control_path` - I set the SSH control path to a directory writable by the user running the script/ansible code (and thus using SSH to connect to the remote server).

The Script

Let's dive right into the main script which uses Ansible:

```
from ansible.playbook import PlayBook
from ansible.inventory import Inventory
from ansible import callbacks
from ansible import utils

import jinja2
from tempfile import NamedTemporaryFile
import os

# Boilerplace callbacks for stdout/stderr and log output
utils.VERBOSITY = 0
playbook_cb = callbacks.PlaybookCallbacks(verbose=utils.VERBOSITY)
stats = callbacks.AggregateStats()
runner_cb = callbacks.PlaybookRunnerCallbacks(stats, verbose=utils.VERBOSITY)

# Dynamic Inventory
# We fake a inventory file and let Ansible load if it's a real file.
# Just don't tell Ansible that, so we don't hurt its feelings.
inventory = """
[customer]
{{ public_ip_address }}

[customer:vars]
domain={{ domain_name }}
customer_id={{ customer_id }}
customer_name={{ customer_name }}
customer_email={{ customer_email }}
"""

inventory_template = jinja2.Template(inventory)
rendered_inventory = inventory_template.render({
    'public_ip_address': '111.222.333.444',
    'domain_name': 'some.domainname.com'
    # and the rest of our variables
})

# Create a temporary file and write the template string to it
hosts = NamedTemporaryFile(delete=False)
hosts.write(rendered_inventory)
hosts.close()

pb = PlayBook(
    playbook='/path/to/main/playbook.yml',
    host_list=hosts.name,     # Our hosts, the rendered inventory file
    remote_user='some_user',
    callbacks=playbook_cb,
    runner_callbacks=runner_cb,
    stats=stats,
    private_key_file='/path/to/key.pem'
)

results = pb.run()

# Ensure on_stats callback is called
# for callback modules
playbook_cb.on_stats(pb.stats)

os.remove(hosts.name)

print results
```

Let's cover what's going on in this script:

Some Assumptions

First, I have a directory called `roles` in the same location as this script. However, you can set the `DEFAULT_ROLES_PATH` constant if you have your roles elsewhere.

I don't mention all the roles or playbooks here, as they are the same as you'd use normally!

The playbook file `/path/to/main/playbook.yml` calls on the needed roles. Any variables that may need to be generated are added within the inventory file and used as group variables within the roles.

Lastly, when this code is run, ensure the `ANSIBLE_CONFIG` variable is set with the full path to the `ansible.cfg` path.

Imports

We import all the Ansible items we need. This is fairly easy to follow - follow the imports as file paths in the Ansible Github repository (https://github.com/ansible/ansible), keeping in mind the `__init__.py` file often contains most of the "front-facing" code within a module, such as `ansible.playbook.Playbook` .

I added imports as needed when I "wrote" this script. Credit where credit is due, though - I started with source materials here (http://stackoverflow.com/questions/27590039/running-ansible-playbook-using-python-api/27597987?noredirect=1) and most notably here (http://nbviewer.ipython.org/gist/rdhyee/7011434).

Boiler Plate/Callbacks

```
# Boilerplace callbacks
utils.VERBOSITY = 0
playbook_cb = callbacks.PlaybookCallbacks(verbose=utils.VERBOSITY)
stats = callbacks.AggregateStats()
runner_cb = callbacks.PlaybookRunnerCallbacks(stats, verbose=utils.VERBOSITY)
```

We'll set some required callbacks for Ansible. These are part of their plugin system (You can create your own callbacks). You can see an example Callback plugin skeleton here (https://github.com/ansible/ansible/blob/devel/lib/ansible/callback_plugins/noop.py), which acts as an interface.

These output the status of tasks and the task runner.

The verbosity being set to 0 is the the default output. Each increment (+1) of the verbosity would be the same as using an additional `-v` flag when calling Ansible via CLI (e.g. `ansible -vvv all -m ping` would be a verbosity of 3).

See how that works by searching for VERBOSITY here (https://github.com/ansible/ansible/blob/devel/lib/ansible/utils/__init__.py).

Template

```
# Dynamic Inventory
inventory = """
[customer]
{{ public_ip_address }}

[customer:vars]
# ...
"""

inventory_template = jinja2.Template(inventory)
rendered_inventory = inventory_template.render({ 'variables':'here' })

# Create a temporary file and write the template string to it
hosts = NamedTemporaryFile(delete=False)
hosts.write(rendered_inventory)
hosts.close()
```

We can add inventory into Ansible by creating a string representing an inventory file. The inventory file is normally where we'd define hosts and possibly some host/group variables.

Setting them in a string template like this lets me add them to the plays dynamically. This is easier than writing data to a file and doing checks to ensure there were no write issues (and thus gives us a bit more insurance against running customer's data on the wrong server).

Here we set a template with variables as expected by the Jinja2 template engine. Then we parse the template and get the final string, with variables replaced by their values.

Finally we add that string to a temporary file. That file will be set as the inventory used when Ansible runs.

Running the Playbook

```
pb = PlayBook(
    playbook='/path/to/main/playbook.yml',
    host_list=hosts.name, # Our hosts, the rendered inventory file
    remote_user='some_user',
    callbacks=playbook_cb,
    runner_callbacks=runner_cb,
    stats=stats,
    private_key_file='/path/to/key.pem'
)

results = pb.run()
os.remove(hosts.name)

# Ensure on_stats callback is called
# for callback modules
playbook_cb.on_stats(pb.stats)

print results
```

Lastly we create a `Playbook` object, set our needed information and run it! The playbook arguments should be familiar to anyone who runs Ansible on the CLI. You can see the complete list of available arguments here (https://github.com/ansible/ansible/blob/devel/lib/ansible/playbook/__init__.py).

You can find pretty complete examples of using Ansible commands programmatically within the Github repository (https://github.com/ansible/ansible/tree/devel/bin). This is exactly how the commands (`ansible` , `ansible-playbook` , etc) are used!

> Note that I delete the temporary hosts file to ensure that's not kept around.

Callback Module

We also can create a callback module to get information about how our roles were run. I use this to log the output to a database so I can report on passed Ansible runs.

This relies on the configuration value we set in `ansible.cfg` . The following is file `logger.py` which is in the location we configured Ansible to check for callback modules:

```
import os
import time
import pymysql.cursors  # A pip-installed dependency
from ansible import utils
from ansible.module_utils import basic
from ansible.utils.unicode import to_unicode, to_bytes

# This message will get concatenated to until it's time
# to log "flush" the message to the database
log_message = ''

def banner(msg):
    """Output Trailing Stars"""
    width = 78 - len(msg)
    if width < 3:
        width = 3
    filler = "*" * width
    return "\n%s %s " % (msg, filler)

def append_to_log(msg):
    """Append message to log_message"""
    global log_message
    log_message += msg+"\n"

def flush_to_database(has_errors=False):
    """Save log_message to database"""
    global log_message
    log_type = 'info'

    if has_errors:
        log_type = 'error'

    db = pymysql.connect(host='localhost',
                         user=os.environ['LOCAL_DB_USER'],
                         passwd=os.environ['LOCAL_DB_PASS'],
                         db=os.environ['LOCAL_DB_NAME'],
                         charset='utf8mb4',
                         cursorclass=pymysql.cursors.DictCursor)

    with db.cursor() as cursor:
        sql = "INSERT INTO hosting_logs (type, log, created_at) VALUES (%s, %s, %s)"

        current_time = time.strftime('%Y-%m-%d %H:%M:%S')
        cursor.execute(sql, (
                log_type,
                log_message,
                current_time
            )
        )
        db.commit()

    db.close()

class CallbackModule(object):
    """
    An ansible callback module for saving Ansible output to a database log
    """

    def runner_on_failed(self, host, res, ignore_errors=False):
        results2 = res.copy()
        results2.pop('invocation', None)
        item = results2.get('item', None)

        if item:
            msg = "failed: [%s] => (item=%s) => %s" % (host, item, utils.jsonify(results2))
        else:
            msg = "failed: [%s] => %s" % (host, utils.jsonify(results2))

        append_to_log(msg)

    def runner_on_ok(self, host, res):
        results2 = res.copy()
        results2.pop('invocation', None)

        item = results2.get('item', None)

        changed = results2.get('changed', False)
        ok_or_changed = 'ok'
        if changed:
            ok_or_changed = 'changed'

        msg = "%s: [%s] => (item=%s)" % (ok_or_changed, host, item)

        append_to_log(msg)

    def runner_on_skipped(self, host, item=None):
        if item:
            msg = "skipping: [%s] => (item=%s)" % (host, item)
        else:
            msg = "skipping: [%s]" % host

        append_to_log(msg)

    def runner_on_unreachable(self, host, res):
        item = None

        if type(res) == dict:
            item = res.get('item', None)
            if isinstance(item, unicode):
                item = utils.unicode.to_bytes(item)
            results = basic.json_dict_unicode_to_bytes(res)
        else:
            results = utils.unicode.to_bytes(res)
        host = utils.unicode.to_bytes(host)
        if item:
            msg = "fatal: [%s] => (item=%s) => %s" % (host, item, results)
        else:
            msg = "fatal: [%s] => %s" % (host, results)

        append_to_log(msg)

    def runner_on_no_hosts(self):
        append_to_log("FATAL: no hosts matched or all hosts have already failed -- aborting")
        pass

    def playbook_on_task_start(self, name, is_conditional):
        name = utils.unicode.to_bytes(name)
        msg = "TASK: [%s]" % name
        if is_conditional:
            msg = "NOTIFIED: [%s]" % name

        append_to_log(banner(msg))
    def playbook_on_setup(self):
        append_to_log(banner('GATHERING FACTS'))
        pass

    def playbook_on_play_start(self, name):
        append_to_log(banner("PLAY [%s]" % name))
        pass

    def playbook_on_stats(self, stats):
        """Complete: Flush log to database"""
        has_errors = False
        hosts = stats.processed.keys()

        for h in hosts:
            t = stats.summarize(h)

            if t['failures'] > 0 or t['unreachable'] > 0:
                has_errors = True

            msg = "Host: %s, ok: %d, failures: %d, unreachable: %d, changed: %d, skipped: %d" % (h, t['ok'], t['failure
s'], t['unreachable'], t['changed'], t['skipped'])
            append_to_log(msg)

        flush_to_database(has_errors)
```

That's it! As long as that `logger.py` class is called `CallbackModule`, and it's in the callback modules page, these callback methods will get called.

Resources

- Examples on Running Ansible Programmatically (http://nbviewer.ipython.org/gist/rdhyee/7011434)
- Example usage from the `ansible-playbook` CLI command (https://github.com/ansible/ansible/blob/devel/bin/ansible-playbook#L186). This is *VERY* handy to learn how to run Ansible programmatically, as that's exactly what these commands are doing!
- Use a vault password like so for inventory (https://github.com/ansible/ansible/blob/v1.9.0-2/bin/ansible-playbook#L156) and like so for other use (https://github.com/ansible/ansible/blob/v1.9.0-2/bin/ansible-playbook#L208) (variables).
- Catch Ansible Exceptions like this (https://github.com/ansible/ansible/blob/v1.9.0-2/bin/ansible-playbook#L261-L311)
- Check the result stats to see what happened within each host (https://github.com/ansible/ansible/blob/v1.9.0-2/bin/ansible-playbook#L269-L301)