The Inside Playbook

When localhost isn't what it seems in Red Hat Ansible Automation Platform 2

July 13, 2022 by Phil Griffiths

With Red Hat Ansible Automation Platform 2 and the advent of automation execution environments, some behaviors are now different for. This blog explains the use case around using localhost and options for sharing data and persistent data storage, for VM-based Ansible Automation Platform 2 deployments.

With Ansible Automation Platform 2 and its containerised execution environments, the concept of localhost has altered. Before Ansible Automation Platform 2, you could run a job against localhost, which translated into running on the *underlying tower host*. You could use this to store data and persistent artifacts, although this was not always a good idea or best practice.

Now with Ansible Automation Platform 2, localhost means you're running *inside* a container, which is ephemeral in nature. This means we must do things differently to achieve the same goal. If you consider this a backwards move, think again. In fact, localhost is now no longer tied to a particular host, and with portable execution environments, this means it can run anywhere, with the right environment and software prerequisites already embedded into the execution environment container.

So, if we now have a temporal runtime container and we want to use existing data or persistent data, then what should we do? Let's examine the options for the rest of the article.

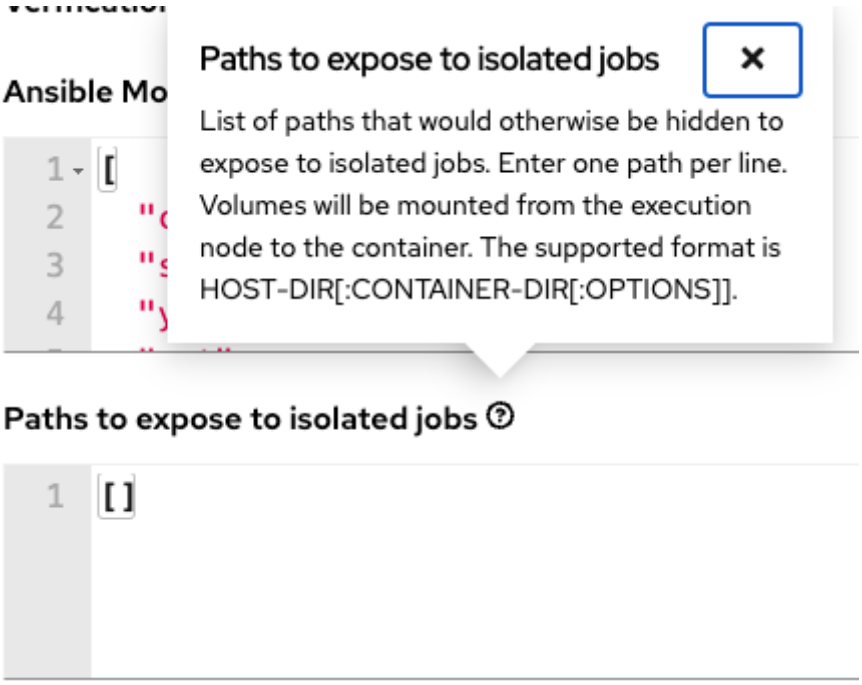
First, we'll look at persisting data from some automation runs.

I would consider using the local Tower filesystem as an anti-pattern, as that could have consequences for platform management and ties the data to that host. If you have a multi-node cluster, then you could hit a different host each time. But just because you can, doesn't mean you should!

So the solution in Ansible Automation Platform 2 is really what we'd suggest for Ansible Automation Platform 1.x as well. Use some form of shared storage solution, like Amazon S3, maybe Gist, or even just have a role to rsync data to your data endpoint. There are of course numerous ways to do this and modules for helping us achieve it.

There is also another option that could prove useful, but has some limitations and caveats, which we'll look at now. At the same time, there is also the use case of injecting some data or configuration into a container at runtime. This can be achieved using automation controller's isolated jobs path option for both needs.

In Ansible Automation Platform 2, under Settings → Jobs → Jobs settings, we have:



This allows us to provide a way to mount directories and files into an execution environment at runtime. This is achieved through the automation mesh, utilizing ansible-runner to inject them into a Podman container to launch the automation. Neat! I have removed the defaults here so we can concentrate on the use case (we'll come back to those defaults later).

So what are some of the use cases for using isolated job paths?

- Providing SSL certificates at runtime, rather than baking them into an execution environment.
- Passing runtime configuration data, such as SSH config settings, but could be anything you want to provide and use during automation.
- Reading and writing to files used before, during and after automation runs.

As I mentioned, there are some caveats to be aware of:

- The volume mount **has to pre-exist** on all nodes capable of automation execution (so hybrid control plane nodes and all execution nodes).
- Where **SELinux** is enabled (Ansible Automation Platform default) beware of file permissions!
    - Especially since we also run **rootless podman** on non-OCP based installs.

The last two points need to be carefully observed, so I highly recommend reading up on rootless Podman and the Podman volume mount runtime options (the *[:OPTIONS]* part of the isolated job paths) as this is what we use inside Ansible Automation Platform 2. I shall demonstrate.

Examples

Let's examine a few of these options to highlight some use cases.

Let's say I have a shared directory called "*/mydata*" in which I want to be able to read and write to files during a job run. Remember this has to already exist on the execution node I'll be using for the automation run.

I'm going to target my *aape1.local* execution node for running this job, so the underlying hosts already has this in place:

```
awx@aape1 ~ $ ls -la /mydata/
total 4
drwxr-xr-x.  2 awx  awx   41 Apr 28 09:27 .
dr-xr-xr-x. 19 root root 258 Apr 11 15:16 ..
-rw-r--r--.  1 awx  awx   33 Apr 11 12:34 file_read
-rw-r--r--.  1 awx  awx    0 Apr 28 09:27 file_write
```

I'm going to use a simple playbook to launch the automation with a sleep in it so we can go in and see what's happening under the covers, as well as demonstrate reading and writing to files.

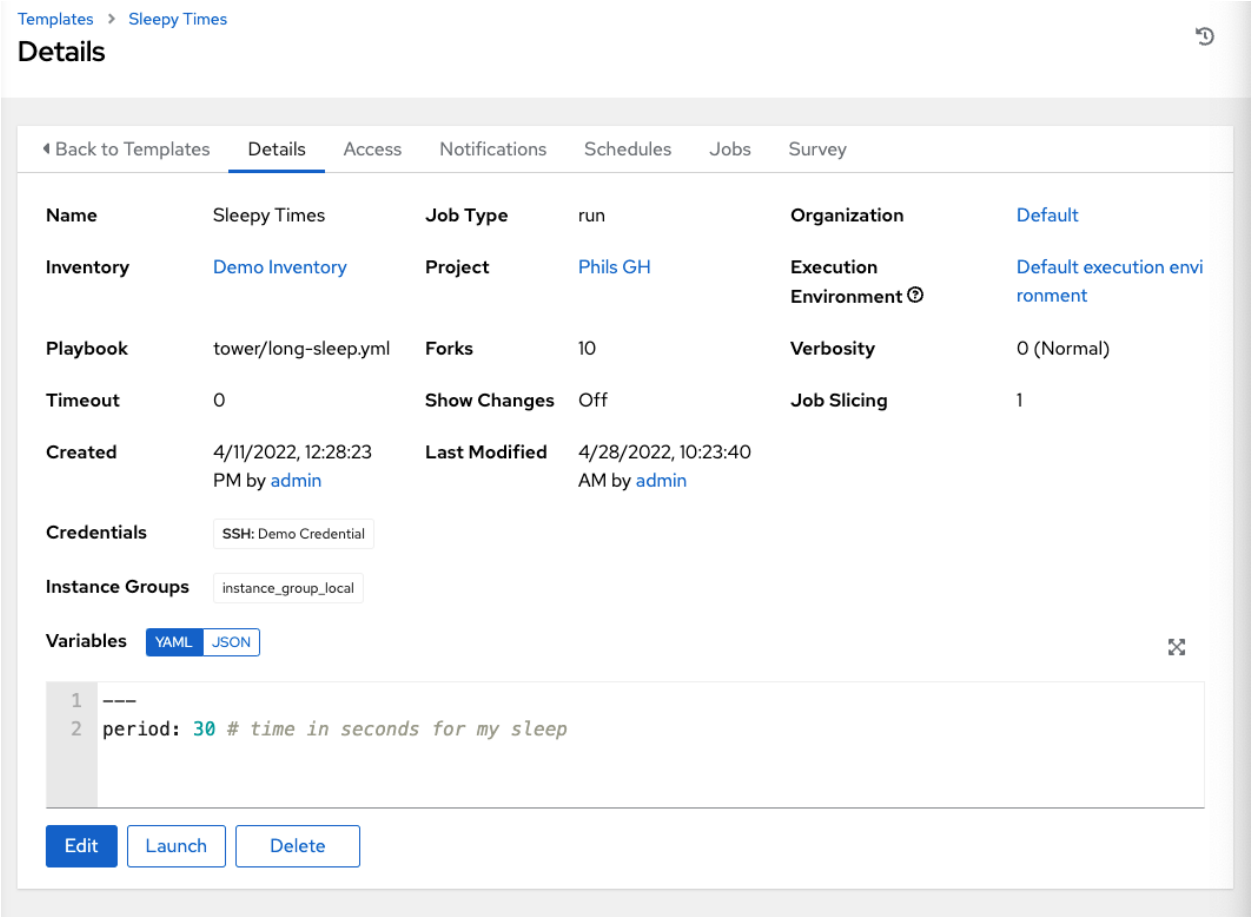
In Ansible Automation Platform 2 under **Settings → Job Settings → Jobs** I've set:

Paths to expose to isolated jobs ⓘ

```
1 [
2   "/mydata:/mydata:rw"
3 ]
```

Here you can see I've mapped the volume mount as the same name into the container and given it read-write capability.

This will now get used when I launch my job template:



I've set *prompt on launch* for **extra_vars** so I can adjust the sleep duration for each run, with a default of 30 seconds.

Once launched, and the *wait_for* module is invoked for the sleep, we can go onto the execution node and look at what's happening.

As it's the only container running, we can run this to get a quick entry into it:

```
[awx@aape1 ~]$ podman exec -it `podman ps -q` /bin/bash
bash-4.4#
```

We are now inside the running execution environment container!

If we look at the permissions, you'll see that **awx** has become 'root', but this isn't really root as in the superuser, as you're using rootless Podman, which maps users into a kernel namespace (think sandbox). If you're interested in how this works, check out the Podman link for shadow-utils.

```
bash-4.4# ls -la /mydata/
total 4
drwxr-xr-x. 2 root root 41 Apr 28 09:27 .
dr-xr-xr-x. 1 root root 77 Apr 28 09:40 ..
-rw-r--r--. 1 root root 33 Apr 11 12:34 file_read
-rw-r--r--. 1 root root  0 Apr 28 09:27 file_write
```

But wait, why has the job failed? We set :rw so it should have write capability, no?

```
TASK [Read pre-existing file...] ********************************************** 10:50:12
ok: [localhost] => {
    "msg": "This is the file I'm reading in."
}

TASK [Write to a new file...] ************************************************** 10:50:12
An exception occurred during task execution. To see the full traceback, use -vvv. The error was:
PermissionError: [Errno 13] Permission denied: b'/mydata/.ansible_tmpazyqyqdrfile_write' -> b'/my
data/file_write'
fatal: [localhost]: FAILED! => {"changed": false, "checksum": "9f576085d584287a3516ee8b3385cc6f69
bf93ce", "msg": "Unable to make b'/root/.ansible/tmp/ansible-tmp-1651139412.9808054-40-9108183438
3738/source' into to /mydata/file_write, failed final rename from b'/mydata/.ansible_tmpazyqyqdrf
ile_write': [Errno 13] Permission denied: b'/mydata/.ansible_tmpazyqyqdrfile_write' -> b'/mydata/
file_write'"}
...ignoring

TASK [Read written out file...] ************************************************ 10:50:13
fatal: [localhost]: FAILED! => {"msg": "An unhandled exception occurred while running the lookup
plugin 'file'. Error was a <class 'ansible.errors.AnsibleError'>, original message: could not loc
ate file in lookup: /mydata/file_write. could not locate file in lookup: /mydata/file_write"}
...ignoring
```

So we were able to read the existing file, but not write out. This is because of SELinux protection. From the Podman docs:

"By default, Podman mounts the volumes in the same mode (read-write or read-only) as it is mounted in the source container. You can change this by adding a ro or rw option. Labeling systems like SELinux require that proper labels are placed on volume content mounted into a container. Without a label, the security system might prevent the processes running inside the container from using the content. ***By default, Podman does not change the labels set by the OS****."*

As we know, this could be a common misinterpretation, leaving you scratching your head. We have set the default to **:z**, which tells Podman to relabel file objects on shared volumes.

So we can either add **:z** or just leave it off:

Paths to expose to isolated jobs

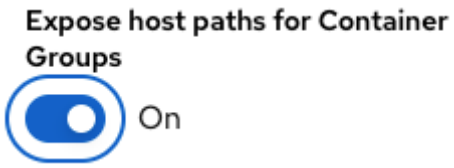
```
[
  "/mydata:/mydata"
]
```

The playbook will now work as expected:

```
PLAY [all] ********************************************************************* 11:05:52

TASK [I'm feeling real sleepy...] ********************************************** 11:05:52
ok: [localhost]

TASK [Read pre-existing file...] *********************************************** 11:05:57
ok: [localhost] => {
    "msg": "This is the file I'm reading in."
}

TASK [Write to a new file...] ************************************************** 11:05:57
ok: [localhost]

TASK [Read written out file...] ************************************************ 11:05:58
ok: [localhost] => {
    "msg": "This is the file I've just written to."
}
```

Back on the underlying execution node host, we have the newly written out contents.

If you're using container groups to launch automation jobs inside Red Hat OpenShift, we can also tell Ansible Automation Platform 2 to expose the same paths to that environment, but we must toggle the default to **On** under settings:



Once enabled, this will inject this as volumeMounts and volumes inside the pod spec that will be used for execution. It'll look something like this (note screenshot straight from docs):

```
apiVersion: v1
kind: Pod
spec:
  containers:
  - image: registry.redhat.io/ansible-automation-platform-22/ee-minimal-rhel8
    args:
      - ansible-runner
      - worker
      - --private-data-dir=/runner
    volumeMounts:
      - mountPath: /mnt2
        name: volume-0
        readOnly: true
      - mountPath: /mnt3
        name: volume-1
        readOnly: true
      - mountPath: /mnt4
        name: volume-2
        readOnly: true
  volumes:
    - hostPath:
        path: /mnt2
        type: ""
      name: volume-0
    - hostPath:
        path: /mnt3
        type: ""
      name: volume-1
    - hostPath:
        path: /mnt4
        type: ""
      name: volume-2
```

Finally, back to the defaults we set:

Paths to expose to isolated jobs ⓘ

```
1 [
2   "/etc/pki/ca-trust:/etc/pki/ca-trust:O",
3   "/usr/share/pki:/usr/share/pki:O"
4 ]
```

These are there so you can include your own SSL certificates at runtime. Place your certificates on all the controller hosts and they will be injected at runtime into the execution environment containers. Note the use of **:O** which tells Podman to mount the directories as temporary storage inside the running container using the overlay file system. Any modifications inside the running container are destroyed after the job completes, much like a tmpfs being unmounted.