
ansible-runner Documentation

Release 2.0.0

Red Hat Ansible

Jun 28, 2021

Contents:

1	Introduction to Ansible Runner	3
1.1	Runner Input Directory Hierarchy	3
1.2	The env directory	4
1.3	env/envvars	4
1.4	env/extravars	4
1.5	env/passwords	4
1.6	env/cmdline	5
1.7	env/ssh_key	5
1.8	env/settings - Settings for Runner itself	5
1.9	Inventory	7
1.10	Project	7
1.11	Modules	7
1.12	Roles	8
1.13	Runner Artifacts Directory Hierarchy	8
1.14	Runner Artifact Job Events (Host and Playbook Events)	9
1.15	Runner Profiling Data Directory	10
2	Installing Ansible Runner	13
2.1	Using pip	13
2.2	Fedora	13
2.3	Debian	13
2.4	From source	14
2.5	Build the distribution	14
2.6	Building the base container image	14
2.7	Building the RPM	14
2.8	Changelog	15
3	Sending Runner Status and Events to External Systems	21
3.1	Event Structure	21
3.2	HTTP Status/Event Emitter Plugin	21
3.3	ZeroMQ Status/Event Emitter Plugin	22
3.4	Writing your own Plugin	22
4	Using Runner as a standalone command line tool	23
4.1	Executing Runner in the foreground	24
4.2	Executing Runner in the background	24
4.3	Running Playbooks	24

4.4	Running Modules Directly	24
4.5	Running Roles Directly	24
4.6	Running with Process Isolation	24
4.7	Running with Directory Isolation	25
4.8	Outputting json (raw event data) to the console instead of normal output	25
4.9	Cleaning up artifact directories	25
5	Using Runner as a Python Module Interface to Ansible	27
5.1	Helper Interfaces	27
5.2	run() helper function	27
5.3	run_async() helper function	27
5.4	run_command() helper function	28
5.5	run_command_async() helper function	28
5.6	get_plugin_docs() helper function	28
5.7	get_plugin_docs_async() helper function	28
5.8	get_plugin_list() helper function	28
5.9	get_inventory() helper function	29
5.10	get_ansible_config() helper function	29
5.11	The Runner object	29
5.12	Runner.stdout	29
5.13	Runner.stderr	30
5.14	Runner.events	30
5.15	Runner.stats	30
5.16	Runner.host_events	30
5.17	Runner.get_fact_cache	30
5.18	Runner.event_handler	30
5.19	Runner.cancel_callback	30
5.20	Runner.finished_callback	30
5.21	Runner.status_handler	31
5.22	Usage examples	31
5.23	Providing custom behavior and inputs	33
6	Using Runner with Execution Environmnets	35
6.1	Emulating the Ansible CLI	35
6.2	Notes and Considerations	36
7	Using Runner as a container interface to Ansible	37
7.1	Overriding the reference container image	37
7.2	Gathering output from the reference container image	37
7.3	Changing the console output to emit raw events	38
8	Remote job execution	39
8.1	Python API	39
9	Indices and tables	41

Ansible Runner is a tool and python library that helps when interfacing with Ansible directly or as part of another system whether that be through a container image interface, as a standalone tool, or as a Python module that can be imported. The goal is to provide a stable and consistent interface abstraction to Ansible. This allows **Ansible** to be embedded into other systems that don't want to manage the complexities of the interface on their own (such as CI/CD platforms, Jenkins, or other automated tooling).

Ansible Runner represents the modularization of the part of [Ansible Tower/AWX](#) that is responsible for running `ansible` and `ansible-playbook` tasks and gathers the output from it. It does this by presenting a common interface that doesn't change, even as **Ansible** itself grows and evolves.

Part of what makes this tooling useful is that it can gather its inputs in a flexible way (See [Introduction to Ansible Runner](#):). It also has a system for storing the output (stdout) and artifacts (host-level event data, fact data, etc) of the playbook run.

There are 3 primary ways of interacting with **Runner**

- A standalone command line tool (`ansible-runner`) that can be started in the foreground or run in the background asynchronously
- A reference container image that can be used as a base for your own images and will work as a standalone container or running in Openshift or Kubernetes
- A python module - library interface

Ansible Runner can also be configured to send status and event data to other systems using a plugin interface, see [Sending Runner Status and Events to External Systems](#).

Examples of this could include:

- Sending status to Ansible Tower/AWX
- Sending events to an external logging service

Introduction to Ansible Runner

Runner is intended to be most useful as part of automation and tooling that needs to invoke Ansible and consume its results. Most of the parameterization of the **Ansible** command line is also available on the **Runner** command line but **Runner** also can rely on an input interface that is mapped onto a directory structure, an example of which can be seen in [the source tree](#).

Further sections in this document refer to the configuration and layout of that hierarchy. This isn't the only way to interface with **Runner** itself. The Python module interface allows supplying these details as direct module parameters in many forms, and the command line interface allows supplying them directly as arguments, mimicking the behavior of `ansible-playbook`. Having the directory structure **does** allow gathering the inputs from elsewhere and preparing them for consumption by **Runner**, then the tooling can come along and inspect the results after the run.

This is best seen in the way Ansible **AWX** uses **Runner** where most of the content comes from the database (and other content-management components) but ultimately needs to be brought together in a single place when launching the **Ansible** task.

1.1 Runner Input Directory Hierarchy

This directory contains all necessary inputs. Here's a view of the [demo directory](#) showing an active configuration.

Note that not everything is required. Defaults will be used or values will be omitted if they are not provided.



(continues on next page)

(continued from previous page)

```
├── test.yml
├── roles
│   └── testrole
│       ├── defaults
│       ├── handlers
│       ├── meta
│       ├── README.md
│       ├── tasks
│       ├── tests
│       └── vars
```

1.2 The `env` directory

The `env` directory contains settings and sensitive files that inform certain aspects of the invocation of the **Ansible** process, an example of which can be found in [the demo env directory](#). Each of these files can also be represented by a named pipe providing a bit of an extra layer of security. The formatting and expectation of these files differs slightly depending on what they are representing.

1.3 `env/envvars`

Note: For an example see [the demo envvars](#).

Ansible Runner will inherit the environment of the launching shell (or container, or system itself). This file (which can be in json or yaml format) represents the environment variables that will be added to the environment at run-time:

```
---
TESTVAR: exampleval
```

1.4 `env/extravars`

Note: For an example see [the demo extravars](#).

Ansible Runner gathers the extra vars provided here and supplies them to the **Ansible Process** itself. This file can be in either json or yaml format:

```
---
ansible_connection: local
test: val
```

1.5 `env/passwords`

Note: For an example see [the demo passwords](#).

Warning: We expect this interface to change/simplify in the future but will guarantee backwards compatibility. The goal is for the user of **Runner** to not have to worry about the format of certain prompts emitted from **Ansible** itself. In particular, vault passwords need to become more flexible.

Ansible itself is set up to emit passwords to certain prompts, these prompts can be requested (`-k` for example to prompt for the connection password). Likewise, prompts can be emitted via `vars_prompt` and also [Ansible Vault](#).

In order for **Runner** to respond with the correct password, it needs to be able to match the prompt and provide the correct password. This is currently supported by providing a yaml or json formatted file with a regular expression and a value to emit, for example:

```
---
"^SSH password:\\s*?$": "some_password"
"^BECOME password.*:\\s*?$": "become_password"
```

1.6 env/cmdline

Warning: Current **Ansible Runner** does not validate the command line arguments passed using this method so it is up to the playbook writer to provide a valid set of options. The command line options provided by this method are lower priority than the ones set by **Ansible Runner**. For instance, this will not override *inventory* or *limit* values.

Ansible Runner gathers command line options provided here as a string and supplies them to the **Ansible Process** itself. This file should contain the arguments to be added, for example:

```
--tags one,two --skip-tags three -u ansible --become
```

1.7 env/ssh_key

Note: Currently only a single ssh key can be provided via this mechanism but this is set to [change soon](#).

This file should contain the ssh private key used to connect to the host(s). **Runner** detects when a private key is provided and will wrap the call to **Ansible** in ssh-agent.

1.8 env/settings - Settings for Runner itself

The **settings** file is a little different than the other files provided in this section in that its contents are meant to control **Runner** directly.

- `idle_timeout: 600` If no output is detected from ansible in this number of seconds the execution will be terminated.
- `job_timeout: 3600` The maximum amount of time to allow the job to run for, exceeding this and the execution will be terminated.
- `pect_timeout: 10` Number of seconds for the internal pexpect command to wait to block on input before continuing

- `pexpect_use_poll`: `True` Use `poll()` function for communication with child processes instead of `select()`. `select()` is used when the value is set to `False`. `select()` has a known limitation of using only up to 1024 file descriptors.
- `suppress_ansi_output`: `False` Allow output from ansible to not be printed to the screen
- `fact_cache`: `'fact_cache'` The directory relative to `artifacts` where `jsonfile` fact caching will be stored. Defaults to `fact_cache`. This is ignored if `fact_cache_type` is different than `jsonfile`.
- `fact_cache_type`: `'jsonfile'` The type of fact cache to use. Defaults to `jsonfile`.

1.8.1 Process Isolation Settings for Runner

The process isolation settings are meant to control the process isolation feature of **Runner**.

- `process_isolation`: `False` Enable limiting what directories on the filesystem the playbook run has access to.
- `process_isolation_executable`: `bwrap` Path to the executable that will be used to provide filesystem isolation.
- `process_isolation_path`: `/tmp` Path that an isolated playbook run will use for staging.
- `process_isolation_hide_paths`: `None` Path or list of paths on the system that should be hidden from the playbook run.
- `process_isolation_show_paths`: `None` Path or list of paths on the system that should be exposed to the playbook run.
- `process_isolation_ro_paths`: `None` Path or list of paths on the system that should be exposed to the playbook run as read-only.

1.8.2 Container-based Execution

The `--containerized` setting instructs **Ansible Runner** to execute **Ansible** tasks inside a container environment. A default execution environment is provided on Docker Hub at [ansible/ansible-runner](https://hub.docker.com/r/ansible/ansible-runner). Users also have the option of building their own container for executing playbooks, however.

To build an execution environment locally, run:

```
docker build --rm=true -t custom-container -f Dockerfile .
```

or, using `podman`:

```
podman build --rm=true -t custom-container -f Dockerfile .
```

To run Ansible Runner with your custom container:

```
ansible-runner run --container-image custom-container -p playbook.yml
```

See `ansible-runner -h` for other container-related options.

1.8.3 Performance Data Collection Settings for Runner

Runner is capable of collecting performance data (namely cpu usage, memory usage, and pid count) during the execution of a playbook run.

Resource profiling is made possible by the use of control groups (often referred to simply as cgroups). When a process runs inside of a cgroup, the resources used by that specific process can be measured.

Before enabling Runner’s resource profiling feature, users must create a cgroup that **Runner** can use. It is worth noting that only privileged users can create cgroups. The new cgroup should be associated with the same user (and related group) that will be invoking **Runner**. The following command accomplishes this on a RHEL system:

```
sudo yum install libcgrouptools
sudo cgcreate -a `whoami` -t `whoami` -g cpuacct,memory,pids:ansible-runner
```

In the above command, `cpuacct`, `memory`, and `pids` refer to kernel resource controllers, while `ansible-runner` refers to the name of the cgroup being created. More detailed information on the structure of cgroups can be found in the RHEL guide on [Managing, monitoring, and updating the kernel](#)

After a cgroup has been created, the following settings can be used to configure resource profiling. Note that `resource_profiling_base_cgroup` must match the name of the cgroup you create.

- `resource_profiling`: `False` Enable performance data collection.
- `resource_profiling_base_cgroup`: `ansible-runner` Top-level cgroup used to measure playbook resource utilization.
- `resource_profiling_cpu_poll_interval`: `0.25` Polling interval in seconds for collecting cpu usage.
- `resource_profiling_memory_poll_interval`: `0.25` Polling interval in seconds for collecting memory usage.
- `resource_profiling_pid_poll_interval`: `0.25` Polling interval in seconds for measuring PID count.
- `resource_profiling_results_dir`: `None` Directory where resource utilization data will be written (if not specified, will be placed in the `profiling_data` folder under the private data directory).

1.9 Inventory

The **Runner** inventory location under the private data dir has the same expectations as inventory provided directly to ansible itself. It can be either a single file or script or a directory containing static inventory files or scripts. This inventory is automatically loaded and provided to **Ansible** when invoked and can be further overridden on the command line or via the `ANSIBLE_INVENTORY` environment variable to specify the hosts directly. Giving an absolute path for the inventory location is best practice, because relative paths are interpreted relative to the `current working directory` which defaults to the `project` directory.

1.10 Project

The **Runner** `project` directory is the playbook root containing playbooks and roles that those playbooks can consume directly. This is also the directory that will be set as the `current working directory` when launching the **Ansible** process.

1.11 Modules

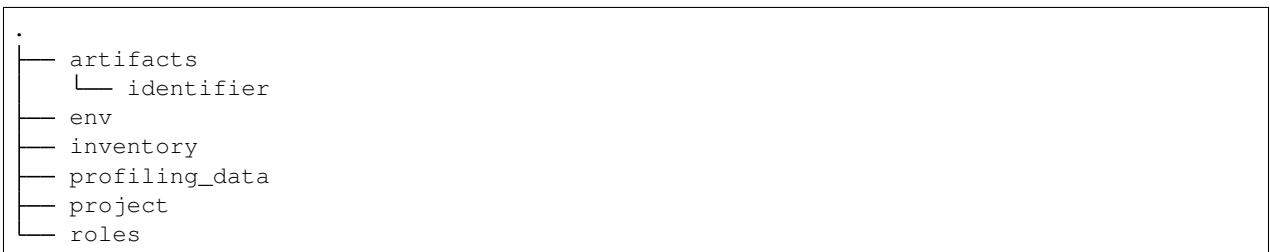
Runner has the ability to execute modules directly using Ansible ad-hoc mode.

1.12 Roles

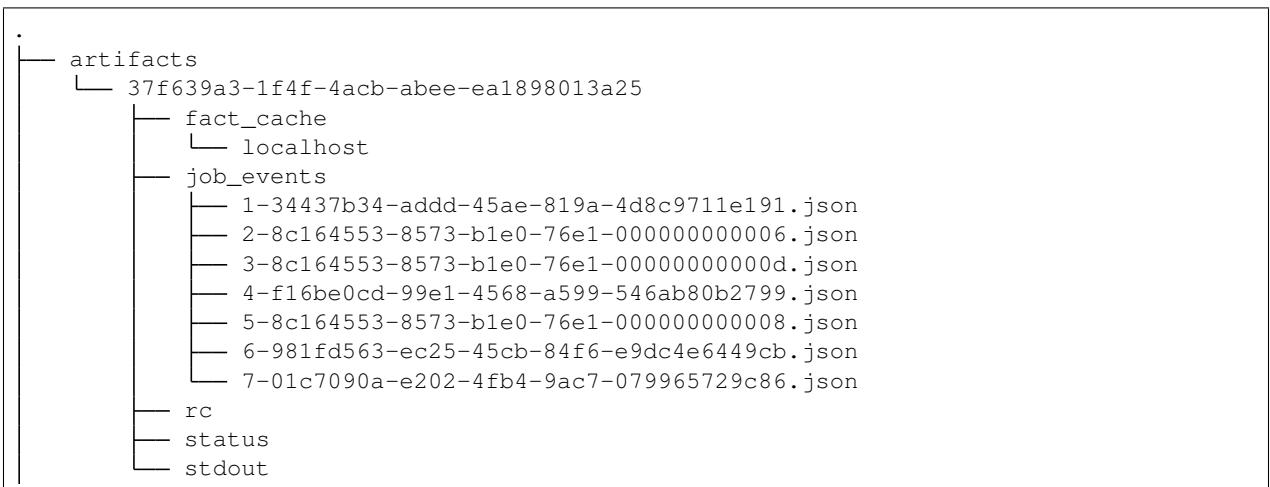
Runner has the ability to execute **Roles** directly without first needing a playbook to reference them. This directory holds roles used for that. Behind the scenes, **Runner** will generate a playbook and invoke the **Role**.

1.13 Runner Artifacts Directory Hierarchy

This directory will contain the results of **Runner** invocation grouped under an `identifier` directory. This identifier can be supplied to **Runner** directly and if not given, an identifier will be generated as a **UUID**. This is how the directory structure looks from the top level:



The artifact directory itself contains a particular structure that provides a lot of extra detail from a running or previously-run invocation of Ansible/Runner:



The **rc** file contains the actual return code from the **Ansible** process.

The **status** file contains one of three statuses suitable for displaying:

- success: The **Ansible** process finished successfully
- failed: The **Ansible** process failed
- timeout: The **Runner** timeout (see *env/settings - Settings for Runner itself*)

The **stdout** file contains the actual stdout as it appears at that moment.

1.14 Runner Artifact Job Events (Host and Playbook Events)

Runner gathers the individual task and playbook events that are emitted as part of the **Ansible** run. This is extremely helpful if you don't want to process or read the stdout returned from **Ansible** as it contains much more detail and status than just the plain stdout. It does some of the heavy lifting of assigning order to the events and stores them in json format under the `job_events` artifact directory. It also takes it a step further than normal **Ansible** callback plugins in that it will store the stdout associated with the event alongside the raw event data (along with stdout line numbers). It also generates dummy events for stdout that didn't have corresponding host event data:

```
{
  "uuid": "8c164553-8573-b1e0-76e1-000000000008",
  "parent_uuid": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxxxx",
  "counter": 5,
  "stdout": "\r\nTASK [debug]\r\n",
  "start_line": 5,
  "end_line": 7,
  "event": "playbook_on_task_start",
  "event_data": {
    "playbook": "test.yml",
    "playbook_uuid": "34437b34-addd-45ae-819a-4d8c9711e191",
    "play": "all",
    "play_uuid": "8c164553-8573-b1e0-76e1-000000000006",
    "play_pattern": "all",
    "task": "debug",
    "task_uuid": "8c164553-8573-b1e0-76e1-000000000008",
    "task_action": "debug",
    "task_path": "\/home\/mjones\/ansible\/ansible-runner\/demo\/project\/test.yml:3",
    "task_args": "msg=Test!",
    "name": "debug",
    "is_conditional": false,
    "pid": 10640
  },
  "pid": 10640,
  "created": "2018-06-07T14:54:58.410605"
}
```

If the playbook runs to completion without getting killed, the last event will always be the `stats` event:

```
{
  "uuid": "01c7090a-e202-4fb4-9ac7-079965729c86",
  "counter": 7,
  "stdout": "\r\nPLAY RECAP\r\n",
  "start_line": 10,
  "end_line": 14,
  "event": "playbook_on_stats",
  "event_data": {
    "playbook": "test.yml",
    "playbook_uuid": "34437b34-addd-45ae-819a-4d8c9711e191",
    "changed": {
      "dark": {

```

(continues on next page)

(continued from previous page)

```

    },
    "failures": {

    },
    "ok": {
        "localhost,": 2
    },
    "processed": {
        "localhost,": 1
    },
    "skipped": {

    },
    "artifact_data": {

    },
    "pid": 10640
},
"pid": 10640,
"created": "2018-06-07T14:54:58.424603"
}

```

Note: The **Runner module interface** presents a programmatic interface to these events that allow getting the final status and performing host filtering of task events.

1.15 Runner Profiling Data Directory

If resource profiling is enabled for **Runner** the `profiling_data` directory will be populated with a set of files containing the profiling data:

```

.
├── profiling_data
│   ├── 0-34437b34-addd-45ae-819a-4d8c9711e191-cpu.json
│   ├── 0-34437b34-addd-45ae-819a-4d8c9711e191-memory.json
│   ├── 0-34437b34-addd-45ae-819a-4d8c9711e191-pids.json
│   ├── 1-8c164553-8573-b1e0-76e1-000000000006-cpu.json
│   ├── 1-8c164553-8573-b1e0-76e1-000000000006-memory.json
│   └── 1-8c164553-8573-b1e0-76e1-000000000006-pids.json

```

Each file is in **JSON text format**. Each line of the file will begin with a record separator (RS), continue with a JSON dictionary, and conclude with a line feed (LF) character. The following provides an example of what the resource files may look like. Note that that since the RS and LF are control characters, they are not actually printed below:

```

==> 0-525400c9-c704-29a6-4107-00000000000c-cpu.json <==
{"timestamp": 1568977988.6844425, "task_name": "Gathering Facts", "task_uuid":
↪ "525400c9-c704-29a6-4107-00000000000c", "value": 97.12799768097156}
{"timestamp": 1568977988.9394386, "task_name": "Gathering Facts", "task_uuid":
↪ "525400c9-c704-29a6-4107-00000000000c", "value": 94.17538298892688}
{"timestamp": 1568977989.1901696, "task_name": "Gathering Facts", "task_uuid":
↪ "525400c9-c704-29a6-4107-00000000000c", "value": 64.38272588006255}
{"timestamp": 1568977989.4594045, "task_name": "Gathering Facts", "task_uuid":
↪ "525400c9-c704-29a6-4107-00000000000c", "value": 83.77387744259856}

```

(continues on next page)

(continued from previous page)

```

==> 0-525400c9-c704-29a6-4107-00000000000c-memory.json <==
{"timestamp": 1568977988.4281094, "task_name": "Gathering Facts", "task_uuid":
↪ "525400c9-c704-29a6-4107-00000000000c", "value": 36.21484375}
{"timestamp": 1568977988.6842303, "task_name": "Gathering Facts", "task_uuid":
↪ "525400c9-c704-29a6-4107-00000000000c", "value": 57.87109375}
{"timestamp": 1568977988.939303, "task_name": "Gathering Facts", "task_uuid":
↪ "525400c9-c704-29a6-4107-00000000000c", "value": 66.60546875}
{"timestamp": 1568977989.1900482, "task_name": "Gathering Facts", "task_uuid":
↪ "525400c9-c704-29a6-4107-00000000000c", "value": 71.4609375}
{"timestamp": 1568977989.4592078, "task_name": "Gathering Facts", "task_uuid":
↪ "525400c9-c704-29a6-4107-00000000000c", "value": 38.25390625}

==> 0-525400c9-c704-29a6-4107-00000000000c-pids.json <==
{"timestamp": 1568977988.4284189, "task_name": "Gathering Facts", "task_uuid":
↪ "525400c9-c704-29a6-4107-00000000000c", "value": 5}
{"timestamp": 1568977988.6845856, "task_name": "Gathering Facts", "task_uuid":
↪ "525400c9-c704-29a6-4107-00000000000c", "value": 6}
{"timestamp": 1568977988.939547, "task_name": "Gathering Facts", "task_uuid":
↪ "525400c9-c704-29a6-4107-00000000000c", "value": 8}
{"timestamp": 1568977989.1902773, "task_name": "Gathering Facts", "task_uuid":
↪ "525400c9-c704-29a6-4107-00000000000c", "value": 13}
{"timestamp": 1568977989.4593227, "task_name": "Gathering Facts", "task_uuid":
↪ "525400c9-c704-29a6-4107-00000000000c", "value": 6}

```

- Resource profiling data is grouped by playbook task.
- For each task, there will be three files, corresponding to cpu, memory and pid count data.
- Each file contains a set of data points collected over the course of a playbook task.
- If a task executes quickly and the polling rate for a given metric is large enough, it is possible that no profiling data may be collected during the task's execution. If this is the case, no data file will be created.

CHAPTER 2

Installing Ansible Runner

Ansible Runner is provided from several different locations depending on how you want to use it.

2.1 Using pip

Python 2.7+ and 3.6+ are supported and installable via pip:

```
$ pip install ansible-runner
```

2.2 Fedora

To install from the latest Fedora sources:

```
$ dnf install python-ansible-runner
```

2.3 Debian

Add an ansible-runner repository:

```
$ apt-get update
$ echo 'deb https://releases.ansible.com/ansible-runner/deb/ <trusty|xenial|stretch>_
↪main' > /etc/apt/sources.list.d/ansible.list
```

Add a key:

```
$ apt-key adv --keyserver keyserver.ubuntu.com --recv 3DD29021
```

Install the package:

```
$ apt-get update
$ apt-get install ansible-runner
```

2.4 From source

Check out the source code from [github](#):

```
$ git clone git://github.com/ansible/ansible-runner
```

Or download from the [releases page](#)

Then install:

```
$ pip install virtualenvwrapper
$ mkvirtualenv ansible-runner
$ pip install -e .
```

2.5 Build the distribution

To produce both wheel and sdist:

```
make dist
```

To produce an installable wheel:

```
make wheel
```

To produce a distribution tarball:

```
make sdist
```

2.6 Building the base container image

Make sure the wheel distribution is built (see *Build the distribution*) and run:

```
make image
```

2.7 Building the RPM

The RPM build uses a container image to bootstrap the environment in order to produce the RPM. Make sure you have docker installed and proceed with:

```
make rpm
```

2.8 Changelog

2.8.1 2.0.0a3 (2020-05-20)

- Updated streaming module to fail jobs when a project contains an invalid symlink in the source files

2.8.2 1.4.7 (2021-02-03)

- Fixed a bug that broke Runner when a bad utf-8 byte sequence was encountered from a stdin/stdout/stderr stream

2.8.3 1.4.6 (2020-03-26)

- Fixed a bug that broke Ansible playbook execution prior to version 2.8 of Ansible.

2.8.4 1.4.5 (2020-03-19)

- Fix an issue with `--process_isolation_*_paths` parsing cli args
- Switch default docker images to centos:8
- Switch extravar format so we can support more than just string types
- Make sure job events folder is created earlier to prevent errors when using immediately after starting a runner job
- Annotate all `runner_on` events with start/end/duration times

2.8.5 1.4.4 (2019-10-25)

- Fix some unicode issues when using command line override on python 2.7
- Fix an issue with file handles not closing on the pexpect pty
- Fix missing `ssh_key` parameter from module interface
- Fix a bug where the ssh agent process would hang around after process isolation exit causing a failure to remove temporary directories

2.8.6 1.4.2 (2019-10-04)

- Reverted `ansible-runner --version` to print a semantic version number

2.8.7 1.4.1 (2019-10-03)

- Fixed a bug that prevented `ANSIBLE_HOST_KEY_CHECKING` from being respected

2.8.8 1.4.0 (2019-09-20)

- Added changed count to stats data
- Added initial support for gathering performance statistics using the system's cgroup interface
- Fix command line args override missing from module run kwargs
- Omit inventory argument entirely if no inventory content is supplied this allows ansible to pick up inventory from implicit locations and ansible.cfg
- Fix an issue where Runner wouldn't properly clean up process isolation temporary directories
- Fix error generated if unsafe parameter is used on vars prompt tasks
- Fix an issue where additional callback plugins weren't being used when defined in the environment
- Fix an issue where Runner would stop returning events after the playbook finished when using run_async
- Fix an issue where unicode in task data would cause Runner to fail
- Fix issues using vaulted data that would cause Runner to fail
- Fix an issue where artifact-dir was only allowed in ad-hoc mode

2.8.9 1.3.4 (2019-04-25)

- Removed an explicit version pin of the six library (which is unavailable in certain Linux distributions).
- Fixed an event handling bug in the callback plugin in Ansible2.9+

2.8.10 1.3.3 (2019-04-22)

- Fix various issues involving unicode input and output
- Fix an issue where cancelling execution could cause an error rather than assigning the proper return code and exiting cleanly
- Fix various errors that would cause Runner to silently exit if some dependencies weren't met or some commands weren't available
- Fix an issue where the job_events directory wasn't created and would result in no output for non-ansible commands

2.8.11 1.3.2 (2019-04-10)

- Add direct support for forks and environment variable in parameterization
- Fix a bug where unicode in playbooks would cause a crash
- Fix a bug where unicode in environment variables would cause a crash
- Capture command and cwd as part of the artifacts delivered for the job
- Automatically remove process isolation temp directories
- Fail more gracefully if ansible and/or bubblewrap isn't available at startup
- Fix an issue where *verbose* events would be delayed until the end of execution

2.8.12 1.3.1 (2019-03-27)

- Fixes to make default file permissions much more secure (0600)
- Adding git to the reference container image to support galaxy requests

2.8.13 1.3.0 (2019-03-20)

- Add support for directory isolation
- Add Debian packaging support
- Add fact caching support
- Add process isolation configuration in the settings file
- Fix event and display issues related to alternative Ansible strategies
- Add Runner config reference to status handler callback
- Add some more direct access to various ansible command line arguments
- Adding playbook stats for “ignored” and “rescued”
- Fix loading of some ansible resources from outside of the private data directory (such as projects/playbooks)
- Fix handling of artifact dir when specified outside of the private data directory
- Fix an issue where the stdout handle wasn’t closed and not all data would be flushed
- Fixed extravar loading behavior
- Added support for resolving parent events by associating their event uuid as parent_uuid
- Allow PYTHONPATH to be overridden
- Expand support for executing non-ansible tools

2.8.14 1.2.0 (2018-12-19)

- Add support for runner_on_start from Ansible 2.8
- Fix thread race condition issues in event gathering
- Add Code Of Conduct
- Fix an issue where the “running” status wouldn’t be emitted to the status callback
- Add process isolation support via bubblewrap
- Fix an issue with orphaned file descriptors
- Add ability to suppress ansible output from the module interface

2.8.15 1.1.2 (2018-10-18)

- Fix an issue where ssh sock path could be too long
- Fix an issue passing extra vars as dictionaries via the interface
- Fix an issue where stdout was delayed on buffering which also caused stdout not to be available if the task was canceled or failed

- Fix role-path parameter not being honored when given on the command line Also fixed up unit tests to actually surface this error if it comes back
- Fully onboard Zuul-CI for unit and integration testing

2.8.16 1.1.1 (2018-09-13)

- Fix an issue when attaching PYTHONPATH environment variable
- Allow selecting a different ansible binary with the RUNNER_BINARY
- Fix `--inventory` command line arguments
- Fix some issues related to terminating ansible
- Add runner ident to to the event processing callback
- Adding integration tests and improving unit tests

2.8.17 1.1.0 (2018-08-16)

- Added a feature that supports sending ansible status and events to external systems via a plugin interface
- Added support for Runner module users to receive runtime status changes in the form of a callback that can be supplied to the `run()` methods (or passing it directly on Runner initialization)
- Fix an issue where timeout settings were far too short
- Add a new status and return code to indicate Runner timeout occurred.
- Add support for running ad-hoc commands (direct module invocation, ala `ansible` vs `ansible-playbook`)
- Fix an issue that caused missing data in events sent to the event handler(s)
- Adding support for supplying `role_path` in module interface
- Fix an issue where messages would still be emitted when `--quiet` was used
- Fix a bug where ansible processes could be orphaned after canceling a job
- Fix a bug where calling the Runner stats method would fail on python 3
- Fix a bug where direct execution of roles couldn't be daemonized
- Fix a bug where relative paths couldn't be used when calling `start` vs `run`

2.8.18 1.0.5 (2018-07-23)

- Fix a bug that could cause a hang if unicode environment variables are used
- Allow `select()` to be used instead of `poll()` when invoking `pepexpect`
- Check for the presence of Ansible before executing
- Fix an issue where a missing project directory would cause Runner to fail silently
- Add support for automatic cleanup/rotation of artifact directories
- Adding support for Runner module users to receive events in the form of a callback that can be supplied to the `run()` methods (or passing it directly on Runner initialization)
- Adding support for Runner module users to provide a callback that will be invoked when the Runner Ansible process has finished. This can be supplied to the `run()` methods (or passing it directly on Runner initialization).

2.8.19 1.0.4 (2018-06-29)

- Adding support for pexpect 4.6 for performance and efficiency improvements
- Adding support for launching roles directly
- Adding support for changing the output mode to json instead of vanilla Ansible (-j)
- Adding arguments to increase ansible verbosity (-v[vvv]) and quiet mode (-q)
- Adding support for overriding the artifact directory location
- Adding the ability to pass arbitrary arguments to the invocation of Ansible
- Improving debug and verbose output
- Various fixes for broken python 2/3 compatibility, including the event generator in the python module
- Fixing a bug when providing an ssh key via the private directory interface
- Fixing bugs that prevented Runner from working on MacOS
- Fixing a bug that caused issues when providing extra vars via the private dir interface

Sending Runner Status and Events to External Systems

Runner can store event and status data locally for retrieval, it can also emit this information via callbacks provided to the module interface.

Alternatively **Runner** can be configured to send events to an external system via installable plugins, there are currently two available

3.1 Event Structure

There are two types of events that are emitted via plugins:

- status events:

These are sent whenever Runner's status changes (see *Runner.status_handler*) for example:

```
{"status": "running", "runner_ident": "XXXX" }
```

- ansible events:

These are sent during playbook execution for every event received from **Ansible** (see *Playbook and Host Events*) for example:

```
{"runner_ident": "XXXX", <rest of event structure > }
```

3.2 HTTP Status/Event Emitter Plugin

This sends status and event data to a URL in the form of json encoded POST requests.

This plugin is available from the [ansible-runner-http github repo](#) and is also available to be installed from pip:

```
$ pip install ansible-runner-http
```

In order to configure it, you can provide details in the Runner Settings file (see [env/settings - Settings for Runner itself](#)):

- `runner_http_url`: The url to receive the `POST`
- `runner_http_headers`: Headers to send along with the request.

The plugin also supports unix file-based sockets with:

- `runner_http_url`: The path to the unix socket
- `runner_http_path`: The path that will be included as part of the request to the socket

Some of these settings are also available as environment variables:

- `RUNNER_HTTP_URL`
- `RUNNER_HTTP_PATH`

3.3 ZeroMQ Status/Event Emitter Plugin

TODO

3.4 Writing your own Plugin

In order to write your own plugin interface and have it be picked up and used by **Runner** there are a few things that you'll need to do.

- Declare the module as a Runner entrypoint in your setup file ([ansible-runner-http](#) has a good example of this):

```
entry_points=({'ansible_runner.plugins': 'modname = your_python_package_name'},
```

- Implement the `status_handler()` and `event_handler()` functions at the top of your package, for example see [ansible-runner-http events.py](#) and the `__init__` import at the top of the module package

After installing this, **Runner** will see the plugin and invoke the functions when status and events are sent. If there are any errors in your plugin they will be raised immediately and **Runner** will fail.

Using Runner as a standalone command line tool

The **Ansible Runner** command line tool can be used as a standard command line interface to **Ansible** itself but is primarily intended to fit into automation and pipeline workflows. Because of this, it has a bit of a different workflow than **Ansible** itself because you can select between a few different modes to launch the command.

While you can launch **Runner** and provide it all of the inputs as arguments to the command line (as you do with **Ansible** itself), there is another interface where inputs are gathered into a single location referred to in the command line parameters as `private_data_dir`. (see [Runner Input Directory Hierarchy](#))

To view the parameters accepted by `ansible-runner`:

```
$ ansible-runner --help
```

An example invocation of the standalone `ansible-runner` utility:

```
$ ansible-runner -p playbook.yml run /tmp/private
```

Where `playbook.yml` is the playbook from the `/tmp/private/projects` directory, and `run` is the command mode you want to invoke **Runner** with

The different **commands** that runner accepts are:

- `run` starts `ansible-runner` in the foreground and waits until the underlying **Ansible** process completes before returning
- `start` starts `ansible-runner` as a background daemon process and generates a pid file
- `stop` terminates an `ansible-runner` process that was launched in the background with `start`
- `is-alive` checks the status of an `ansible-runner` process that was started in the background with `start`
- `ad-hoc` will run ad-hoc `ansible` commands inside a containerized Ansible Execution Environment
- `playbook` will run `ansible-playbook` commands inside a containerized Ansible Execution Environment

While **Runner** is running it creates an `artifacts` directory (see [Runner Artifacts Directory Hierarchy](#)) regardless of what mode it was started in. The resulting output and status from **Ansible** will be located here. You can control the exact location underneath the `artifacts` directory with the `-i IDENT` argument to `ansible-runner`, otherwise a random UUID will be generated.

4.1 Executing Runner in the foreground

When launching **Runner** with the `run` command, as above, the program will stay in the foreground and you'll see output just as you expect from a normal **Ansible** process. **Runner** will still populate the `artifacts` directory, as mentioned in the previous section, to preserve the output and allow processing of the artifacts after exit.

4.2 Executing Runner in the background

When launching **Runner** with the `start` command, the program will generate a pid file and move to the background. You can check its status with the `is-alive` command, or terminate it with the `stop` command. You can find the `stdout`, `status`, and `return code` in the `artifacts` directory.

4.3 Running Playbooks

An example invocation using `demo` as private directory:

```
$ ansible-runner --playbook test.yml run demo
```

4.4 Running Modules Directly

An example invoking the `debug` module with `demo` as a private directory:

```
$ ansible-runner -m debug --hosts localhost -a msg=hello run demo
```

4.5 Running Roles Directly

An example invocation using `demo` as private directory and `localhost` as target:

```
$ ansible-runner --role testrole --hosts localhost run demo
```

Ansible roles directory can be provided with `--roles-path` option. Role variables can be passed with `--role-vars` at runtime.

4.6 Running with Process Isolation

Runner supports process isolation. Process isolation creates a new mount namespace where the root is on a `tmpfs` that is invisible from the host and is automatically cleaned up when the last process exits. You can enable process isolation by providing the `--process-isolation` argument on the command line. **Runner** as of version 2.0 defaults to using `podman` as the process isolation executable, but supports using any executable that is compatible with the `bubblewrap` CLI arguments by passing in the `--process-isolation-executable` argument:

```
$ ansible-runner --process-isolation ...
```

Runner supports various process isolation arguments that allow you to provide configuration details to the process isolation executable. To view the complete list of arguments accepted by `ansible-runner`:

```
$ ansible-runner --help
```

4.7 Running with Directory Isolation

If you need to be able to execute multiple tasks in parallel that might conflict with each other or if you want to make sure a single invocation of Ansible/Runner doesn't pollute or overwrite the playbook content you can give a base path:

```
$ ansible-runner --directory-isolation-base-path /tmp/runner
```

Runner will copy the project directory to a temporary directory created under that path, set it as the working directory, and execute from that location. After running that temp directory will be cleaned up and removed.

4.8 Outputting json (raw event data) to the console instead of normal output

Runner supports outputting json event data structure directly to the console (and stdout file) instead of the standard **Ansible** output, thus mimicing the behavior of the `json` output plugin. This is in addition to the event data that's already present in the artifact directory. All that is needed is to supply the `-j` argument on the command line:

```
$ ansible-runner ... -j ...
```

4.9 Cleaning up artifact directories

Using the command line argument `--rotate-artifacts` allows you to control the number of artifact directories that are present. Given a number as the parameter for this argument will cause **Runner** to clean up old artifact directories. The default value of 0 disables artifact directory cleanup.

Using Runner as a Python Module Interface to Ansible

Ansible Runner is intended to provide a directly importable and usable API for interfacing with **Ansible** itself and exposes a few helper interfaces.

The modules center around the `Runner` object. The helper methods will either return an instance of this object which provides an interface to the results of executing the **Ansible** command or a tuple the actual output and error response based on the interface.

Ansible Runner itself is a wrapper around **Ansible** execution and so adds plugins and interfaces to the system in order to gather extra information and process/store it for use later.

5.1 Helper Interfaces

The helper `interfaces` provides a quick way of supplying the recommended inputs in order to launch a **Runner** process. These interfaces also allow overriding and providing inputs beyond the scope of what the standalone or container interfaces support. You can see a full list of the inputs in the linked module documentation.

5.2 `run()` helper function

```
ansible_runner.interface.run()
```

When called, this function will take the inputs (either provided as direct inputs to the function or from the *Runner Input Directory Hierarchy*), and execute **Ansible**. It will run in the foreground and return the `Runner` object when finished.

5.3 `run_async()` helper function

```
ansible_runner.interface.run_async()
```

Takes the same arguments as `ansible_runner.interface.run()` but will launch **Ansible** asynchronously and return a tuple containing the `thread` object and a `Runner` object. The **Runner** object can be inspected during execution.

5.4 `run_command()` helper function

```
ansible_runner.interface.run_command()
```

When called, this function will take the inputs (either provided as direct inputs to the function or from the *Runner Input Directory Hierarchy*), and execute the command passed either locally or within an container based on the parameters passed. It will run in the foreground and return a tuple of output and error response when finished. While running the within container image command the current local working directory will be volume mounted within the container, in addition to this for any of ansible command line utilities the inventory, vault-password-file, private-key file path will be volume mounted if provided in the `cmdline_args` parameters.

5.5 `run_command_async()` helper function

```
ansible_runner.interface.run_command_async()
```

Takes the same arguments as `ansible_runner.interface.run_command()` but will launch asynchronously and return a tuple containing the `thread` object and a `Runner` object. The **Runner** object can be inspected during execution.

5.6 `get_plugin_docs()` helper function

```
ansible_runner.interface.get_plugin_docs()
```

When called, this function will take the inputs, and execute the `ansible-doc` command to return the either the plugin-docs or playbook snippet for the passed list of plugin names. The plugin docs can be fetched either from locally installed plugins or from within an container image based on the parameters passed. It will run in the foreground and return a tuple of output and error response when finished. While running the command within the container the current local working directory will be volume mounted within the container.

5.7 `get_plugin_docs_async()` helper function

```
ansible_runner.interface.get_plugin_docs_async()
```

Takes the same arguments as `ansible_runner.interface.get_plugin_docs_async()` but will launch asynchronously and return a tuple containing the `thread` object and a `Runner` object. The **Runner** object can be inspected during execution.

5.8 `get_plugin_list()` helper function

```
ansible_runner.interface.get_plugin_list()
```

When called, this function will take the inputs, and execute the `ansible-doc` command to return the list of installed plugins. The installed plugin can be fetched either from local environment or from within an container image based on the parameters passed. It will run in the foreground and return a tuple of output and error response when finished.

While running the command within the container the current local working directory will be volume mounted within the container.

5.9 `get_inventory()` helper function

```
ansible_runner.interface.get_inventory()
```

When called, this function will take the inputs, and execute the `ansible-inventory` command to return the inventory related information based on the action. If *action* is *list* it will return all the applicable configuration options for *ansible*, for *host* action it will return information of a single host and for *graph* action it will return the inventory. The execution will be in the foreground and return a tuple of output and error response when finished. While running the command within the container the current local working directory will be volume mounted within the container.

5.10 `get_ansible_config()` helper function

```
ansible_runner.interface.get_ansible_config()
```

When called, this function will take the inputs, and execute the `ansible-config` command to return the Ansible configuration related information based on the action. If *action* is *list* it will return all the hosts related information including the host and group variables, for *dump* action it will return the entire active configuration and it can be customized to return only the changed configuration value by setting the *only_changed* boolean parameter to *True*. For *view* action it will return the view of the active configuration file. The execution will be in the foreground and return a tuple of output and error response when finished. While running the command within the container the current local working directory will be volume mounted within the container.

5.11 The Runner object

The `Runner` object is returned as part of the execution of **Ansible** itself. Since it wraps both execution and output it has some helper methods for inspecting the results. Other than the methods and indirect properties, the instance of the object itself contains two direct properties:

- `rc` will represent the actual return code of the **Ansible** process
- **status** will represent the state and can be one of:
 - `unstarted`: This is a very brief state where the `Runner` task has been created but hasn't actually started yet.
 - `successful`: The `ansible` process finished successfully.
 - `failed`: The `ansible` process failed.

5.12 `Runner.stdout`

The `Runner` object contains a property `ansible_runner.runner.Runner.stdout` which will return an open file handle containing the `stdout` of the **Ansible** process.

5.13 Runner.stderr

When the `runner_mode` is set to `subprocess` the `Runner` object uses a property `ansible_runner.runner.Runner.stderr` which will return an open file handle containing the `stderr` of the **Ansible** process.

5.14 Runner.events

`ansible_runner.runner.Runner.events` is a generator that will return the *Playbook and Host Events* as Python dict objects.

5.15 Runner.stats

`ansible_runner.runner.Runner.stats` is a property that will return the final `playbook stats` event from **Ansible** in the form of a Python dict

5.16 Runner.host_events

`ansible_runner.runner.Runner.host_events()` is a method that, given a hostname, will return a list of only **Ansible** event data executed on that Host.

5.17 Runner.get_fact_cache

`ansible_runner.runner.Runner.get_fact_cache()` is a method that, given a hostname, will return a dictionary containing the **Facts** stored for that host during execution.

5.18 Runner.event_handler

A function passed to `__init__` of `Runner`, this is invoked every time an Ansible event is received. You can use this to inspect/process/handle events as they come out of Ansible. This function should return `True` to keep the event, otherwise it will be discarded.

5.19 Runner.cancel_callback

A function passed to `__init__` of `Runner`, and to the `ansible_runner.interface.run()` interface functions. This function will be called for every iteration of the `ansible_runner.interface.run()` event loop and should return `True` to inform **Runner** cancel and shutdown the **Ansible** process or `False` to allow it to continue.

5.20 Runner.finished_callback

A function passed to `__init__` of `Runner`, and to the `ansible_runner.interface.run()` interface functions. This function will be called immediately before the **Runner** event loop finishes once **Ansible** has been shut down.

5.21 Runner.status_handler

A function passed to `__init__` of `Runner` and to the `ansible_runner.interface.run()` interface functions. This function will be called any time the status changes, expected values are:

- *starting*: Preparing to start but hasn't started running yet
- *running*: The **Ansible** task is running
- *canceled*: The task was manually canceled either via callback or the cli
- *timeout*: The timeout configured in Runner Settings was reached (see *env/settings - Settings for Runner itself*)
- *failed*: The **Ansible** process failed

5.22 Usage examples

```
import ansible_runner
r = ansible_runner.run(private_data_dir='/tmp/demo', playbook='test.yml')
print("{}: {}".format(r.status, r.rc))
# successful: 0
for each_host_event in r.events:
    print(each_host_event['event'])
print("Final status:")
print(r.stats)
```

```
import ansible_runner
r = ansible_runner.run(private_data_dir='/tmp/demo', host_pattern='localhost', module=
→ 'shell', module_args='whoami')
print("{}: {}".format(r.status, r.rc))
# successful: 0
for each_host_event in r.events:
    print(each_host_event['event'])
print("Final status:")
print(r.stats)
```

```
# run ansible/generic commands in interactive mode within container
out, err = ansible_runner.run_command(
    executable_cmd='ansible-playbook',
    cmdline_args=['gather.yml', '-i', 'inventory', '-vvvv', '-k'],
    input_fd=sys.stdin,
    output_fd=sys.stdout,
    error_fd=sys.stderr,
    host_cwd='/home/demo',
    process_isolation=True,
    container_image='network-ee'
)
print("out: {}".format(out))
print("err: {}".format(err))
```

```
# run ansible/generic commands in interactive mode locally
out, err = ansible_runner.run_command(
    executable_cmd='ansible-playbook',
    cmdline_args=['gather.yml', '-i', 'inventory', '-vvvv', '-k'],
    input_fd=sys.stdin,
```

(continues on next page)

(continued from previous page)

```

        output_fd=sys.stdout,
        error_fd=sys.stderr,
    )
    print("out: {}".format(out))
    print("err: {}".format(err))

```

```

# get plugin docs from within container
out, err = ansible_runner.get_plugin_docs(
    plugin_names=['vyos.vyos.vyos_command'],
    plugin_type='module',
    response_format='json',
    process_isolation=True,
    container_image='network-ee'
)
print("out: {}".format(out))
print("err: {}".format(err))

```

```

# get plugin docs from within container in async mode
thread_obj, runner_obj = ansible_runner.get_plugin_docs_async(
    plugin_names=['ansible.netcommon.cli_config', 'ansible.netcommon.cli_command'],
    plugin_type='module',
    response_format='json',
    process_isolation=True,
    container_image='network-ee'
)
while runner_obj.status not in ['canceled', 'successful', 'timeout', 'failed']:
    time.sleep(0.01)
    continue

print("out: {}".format(runner_obj.stdout.read()))
print("err: {}".format(runner_obj.stderr.read()))

```

```

# get plugin list installed on local system
out, err = ansible_runner.get_plugin_list()
print("out: {}".format(out))
print("err: {}".format(err))

```

```

# get plugins with file list from within container
out, err = ansible_runner.get_plugin_list(list_files=True, process_isolation=True,
    ↪container_image='network-ee')
print("out: {}".format(out))
print("err: {}".format(err))

```

```

# get list of changed ansible configuration values
out, err = ansible_runner.get_ansible_config(action='dump', config_file='/home/demo/
    ↪ansible.cfg', only_changed=True)
print("out: {}".format(out))
print("err: {}".format(err))

# get ansible inventory information
out, err = ansible_runner.get_inventory(
    action='list',
    inventories=['/home/demo/inventory1', '/home/demo/inventory2'],
    response_format='json',
    process_isolation=True,

```

(continues on next page)

(continued from previous page)

```
        container_image='network-ee'
    )
    print("out: {}".format(out))
    print("err: {}".format(err))
```

5.23 Providing custom behavior and inputs

TODO

The helper methods are just one possible entrypoint, extending the classes used by these helper methods can allow a lot more custom behavior and functionality.

Show:

- How `Runner Config` is used and how overriding the methods and behavior can work
- Show how custom cancel and status callbacks can be supplied.

Using Runner with Execution Environmnets

Execution Environments are meant to be a consistent, reproducible, portable, and sharable method to run Ansible Automation jobs in the exact same way on your laptop as they are executed in ‘**Ansible AWX**’ <<https://github.com/ansible/awx/>>. This aids in the development of automation jobs and Ansible Content that is meant to be run in **Ansible AWX**, **Ansible Tower**, or via **Red Hat Ansible Automation Platform** in a predictable way.

More specifically, the term **Execution Environments** within the context of **Ansible Runner** refers to the container runtime execution of **Ansible** via **Ansible Runner** within an **OCI Compliant Container Runtime** using an **OCI Compliant Container Image** that appropriately bundles **Ansible Base**, **Ansible Collection Content**, and the runtime dependencies required to support these contents. The base image is the **Red Hat Enterprise Linux Universal Base Image** and the build tooling provided by **Ansible Builder** aids in the creation of these images.

All aspects of running **Ansible Runner** in standalone mode (see: *Using Runner as a standalone command line tool*) are true here with the exception that the process isolation is inherently a container runtime (**podman** by default).

6.1 Emulating the Ansible CLI

As previously mentioned, a primary goal of adding the Execution Environment CLI interface is to aid in the creation of Ansible Automation jobs and content. The approach here is to make it as similar as possible to the way **Ansible** users are accustomed to using Ansible today. There are two subcommands, **adhoc** and **playbook** that have been added to accommodate this. The **adhoc** subcommand to **ansible-runner** is synonymous with **ansible** and the **playbook** subcommand to **ansible-runner** is synonymous with **ansible-playbook**. Examples are below.

6.1.1 Running Ansible adhoc

An example invocation using the **ping** module and **localhost** as target:

```
$ ansible-runner adhoc localhost -m ping
```

Something to note here is that implicit **localhost** in this context is a containerized instantiation of an Ansible Execution Environment and as such you will not get Ansible Facts about your system if using the **setup** module.

6.1.2 Running Ansible `ansible-playbook`

An example invocation using the `demo.yml` playbook and `inventory.ini` inventory file:

```
$ ansible-runner playbook demo.yml -i inventory.ini
```

Something to note here is that implicit `localhost` in this context is a containerized instantiation of an Ansible Execution Environment and as such you will not get Ansible Facts about your system if using `gather_facts: true` and targeting `localhost` in your playbook without explicit host definition in your inventory.

6.2 Notes and Considerations

There are some differences between using Ansible Runner and running Ansible directly from the command line that have to do with configuration, content locality, and secret data.

6.2.1 Secrets

Typically with Ansible you are able to provide secret data via a series of mechanisms, many of which are pluggable and configurable. When using Ansible Runner, however, certain considerations need to be made; these are analogous to how Ansible AWX and Tower manage this information.

See *[Runner Input Directory Hierarchy](#)* for more information

6.2.2 Container Names

Like all `ansible-runner` jobs, each job has an identifier associated with it which is also the name of the artifacts subfolder where results are saved to. When a container for job isolation is launched, it will be given a name of `ansible_runner_<job identifier>`. Some characters from the job identifier may be replaced with underscores for compatibility with names that Podman and Docker allow.

This name is used internally if a command needs to be ran against the container at a later time (e.g., to stop the container when the job is canceled).

6.2.3 `~/.ssh/` symlinks

In order to make the `adhoc` and `playbook` container execution of Ansible easier, Ansible Runner will automatically bind mount your local `ssh` agent UNIX-domain socket (`SSH_AUTH_SOCK`) into the container runtime. However, this does not work if files in your `~/.ssh/` directory happen to be symlinked to another directory that is also not mounted into the container runtime. Ansible Runner `adhoc` and `playbook` subcommands provide the `--container-volume-mount` option to address this, among other things.

Here is an example of an `ssh` config file that is a symlink:

```
$ $ ls -l ~/.ssh/config
lrwxrwxrwx. 1 myuser myuser 34 Jul 15 19:27 /home/myuser/.ssh/config -> /home/myuser/
↳ dotfiles/ssh_config

$ ansible-runner playbook \
  --container-volume-mount /home/myuser/dotfiles:/home/myuser/dotfiles/ \
  my_playbook.yml -i my_inventory.ini
```

Using Runner as a container interface to Ansible

The design of **Ansible Runner** makes it especially suitable for controlling the execution of **Ansible** from within a container for single-purpose automation workflows. A reference container image definition is [provided](#) and is also published to [DockerHub](#) you can try it out for yourself

```
$ docker run --rm -e RUNNER_PLAYBOOK=test.yml -v $PWD/demo:/runner quay.io/ansible/
↪ansible-runner:latest
PLAY [all] *****

TASK [Gathering Facts] *****
ok: [localhost]

TASK [debug] *****
ok: [localhost] => {
  "msg": "Test!"
}

PLAY RECAP *****
localhost                : ok=2    changed=0    unreachable=0    failed=0
```

The reference container image is purposefully light-weight and only containing the dependencies necessary to run `ansible-runner` itself. It's intended to be overridden.

7.1 Overriding the reference container image

TODO

7.2 Gathering output from the reference container image

TODO

7.3 Changing the console output to emit raw events

This can be useful when directing task-level event data to an external system by means of the container's console output.

See *Running with Process Isolation*

Remote job execution

Ansible Runner supports the concept that a job run may be requested on one host but executed on another. This capability is primarily intended to be used by [Receptor](#).

Support for this in Runner involves a three phase process.

- **Transmit:** Convert the job to a binary format that can be sent to the worker node.
- **Worker:** Actually execute the job.
- **Process:** Receive job results and process them.

The following command illustrates how the three phases work together:

```
$ ansible-runner transmit ./demo -p test.yml | ansible-runner worker | ansible-runner process ./demo
```

In this example, the *ansible-runner transmit* command is given a private data directory of *./demo* and told to select the *test.yml* playbook from it. Instead of executing the playbook as *ansible-runner run* would do, the data dir and command line parameters are converted to a compressed binary stream that is emitted as stdout. The *transmit* command generally takes the same command line parameters as the *run* command.

The *ansible-runner worker* command accepts this stream, runs the playbook, and generates a new compressed binary stream of the resulting job events and artifacts. This command optionally accepts the *-private-data-dir* option. If provided, it will extract the contents sent from *ansible-runner transmit* into that directory. If no *-private-data-dir* is given, then it will extract the contents to a temporary directory, which will be deleted at the end of execution.

The *ansible-runner process* command accepts the result stream from the worker, and fires all the normal callbacks and does job event processing. In the command above, this results in printing the playbook output and saving artifacts to the data dir. The *process* command takes a data dir as a parameter, to know where to save artifacts.

8.1 Python API

Python code importing Ansible Runner can make use of these facilities by setting the *streamer* parameter to *ansible_runner.interface.run*. This parameter can be set to *transmit*, *worker* or *process* to invoke each of the three stages.

Other parameters are as normal in the CLI.

CHAPTER 9

Indices and tables

- `genindex`
- `modindex`
- `search`