ansible-runner Documentation

Release 1.1.1

Red Hat Ansible

Sep 13, 2018

Contents:

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

Contents:

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.

```
.
├── env
│   ├── envvars
│   ├── extravars
│   ├── passwords
│   ├── cmdline
│   ├── settings
│   └── ssh_key
├── inventory
│   └── hosts
├── project
```
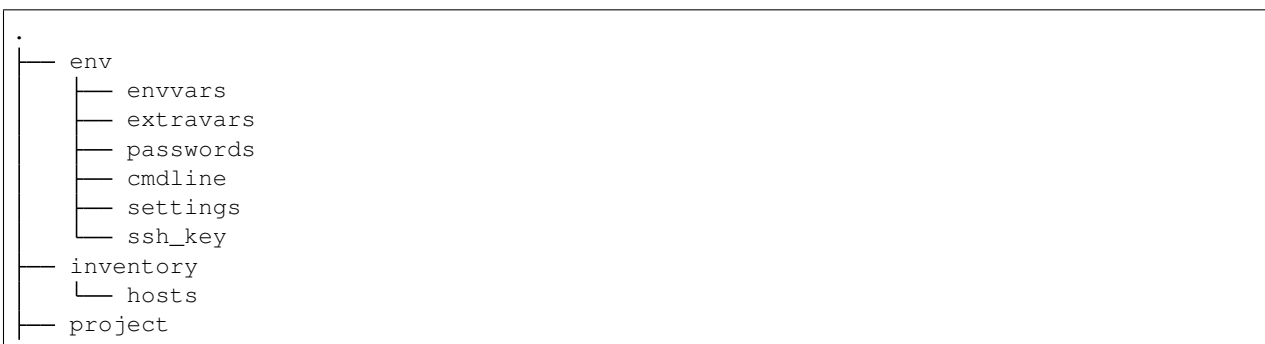
```
    │       └── test.yml
    └── roles
        └── testrole
            ├── defaults
            ├── handlers
            ├── meta
            ├── README.md
            ├── tasks
            ├── tests
            └── vars
```

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 [pP]assword:$": "some_password"
"^BECOME [pP]assword:$": "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.
pexpect_timeout: 10 # Number of seconds for the internal pexpect command to wait to
→block on input before continuing
```

(continues on next page)

```
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_ansible_output: False # Allow output from ansible to not be printed to the␣
↪screen
```

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 limited or overridden on the command line or via an environment variable to specify the hosts directly.

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:

```
.
├── artifacts
│   └── identifier
├── env
├── inventory
├── project
└── roles
```

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:

```
.
├── artifacts
│   └── 37f639a3-1f4f-4acb-abee-ea1898013a25
│       ├── job_events
│       │   ├── 1-34437b34-addd-45ae-819a-4d8c9711e191.json
│       │   ├── 2-8c164553-8573-b1e0-76e1-000000000006.json
│       │   ├── 3-8c164553-8573-b1e0-76e1-00000000000d.json
│       │   ├── 4-f16be0cd-99e1-4568-a599-546ab80b2799.json
│       │   ├── 5-8c164553-8573-b1e0-76e1-000000000008.json
│       │   ├── 6-981fd563-ec25-45cb-84f6-e9dc4e6449cb.json
│       │   └── 7-01c7090a-e202-4fb4-9ac7-079965729c86.json
│       ├── rc
│       ├── status
│       └── stdout
```

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",
  "counter": 5,
  "stdout": "\r\nTASK [debug]␣
→****************************************************************",
  "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",
```

(continues on next page)

```
    "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\u001b[0;
→32mlocalhost,\u001b[0m                     : \u001b[0;32mok=2    \u001b[0m changed=0   ␣
→unreachable=0    failed=0   \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": {

    },
    "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.

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 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:

```
$ python setup.py install
```

OR:

```
$ pip install .
```

2.4 Build the distribution

To produce an installable `wheel` file:

```
make dist
```

To produce a distribution tarball:

```
make sdist
```

2.5 Building the base container image

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

```
make image
```

2.6 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
```

2.7 Changelog

2.7.1 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

2.7.2 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

2.7.3 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 pexpect

- 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).

2.7.4 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`

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 invocating 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 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 mimicking 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 ...
```

4.7 Cleaning up artifact directories

Using the command line argument `--runner-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 return an instance of this object which provides an interface to the results of executing the **Ansible** command.

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.

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 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.

5.5 `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.

5.6 `Runner.events`

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

5.7 `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`

`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.

5.8 `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.

5.9 `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.

5.10 `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.

5.11 `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

5.12 Usage examples

```python
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)
```

```python
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)
```

5.13 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 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 ansible/ansible-runner:latest
  Unable to find image 'ansible/ansible-runner:latest' locally
  latest: Pulling from ansible/ansible-runner
  [...]
  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.

6.1 Overriding the reference container image

TODO

6.2 Gathering output from the reference container image

TODO

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

CHAPTER 7

Indices and tables

- genindex
- modindex
- search