ansible-builder Documentation

Red Hat Ansible

Oct 27, 2025

CONTENTS:

With `ansible-builder` you can configure and build portable, consistent, customized Ansible control nodes that are packaged as containers by Podman or Docker. These containers are known as execution environments. You can use them on AWX or Ansible Controller, with Ansible Navigator, for local playbook development and testing, in your CI pipelines, and anywhere else you run automation.

You can design and distribute specialized execution environments for your Ansible content, choosing the versions of Python and ansible-core you want, and installing only the Python packages, system packages, and Ansible collections you need for your tasks.

CHAPTER

ONE

CONTAINER CONCEPTS AND TERMS

Ansible Builder depends on more generalized containerization tools like Podman or Docker.

Before you start using Ansible Builder, you should understand the following concepts and terms relevant to any use of containers:

- **Build instruction file** (called a `Containerfile` in Podman and a `Dockerfile` in Docker): an instruction file for creating a container image by installing and configuring the code and dependencies.
- **Container**: a package of code and dependencies that runs a service or an application across a variety of computing environments.
- **Image**: a complete but inactive version of a container - you can distribute images and create one or more containers based on each image.

CHAPTER

TWO

WHAT ARE EXECUTION ENVIRONMENTS?

Refer to the Getting started with Execution Environments guide for details.

CHAPTER

THREE

QUICKSTART FOR ANSIBLE BUILDER

To get started with Ansible Builder, you must install the `ansible-builder` utility and a containerization tool.

Once you have the tools you need, create an *execution environment definition* file. By default, this file is called `execution-environment.yml` (the `.yaml` extension is also accepted). In the execution environment definition file, you can specify the exact content you want to include in your execution environment. You can specify these items:

- the base container image
- the version of Python
- the version of ansible-core
- the version of ansible-runner
- Ansible collections, with version restrictions
- system packages, with version restrictions
- Python packages, with version restrictions
- other items to download, install, or configure

3.1 Choosing a base image

You can use any base image you choose. The smaller the base image, generally, the smaller the final image. However, to make Ansible Builder more efficient, you should know what packages, if any, are already installed on the base image you use.

For example, some base images already have Python installed. Others do not. If you use a base image that already has Python installed, you can omit Python in your execution environment definition file. Not all base images have package managers installed.

CHAPTER

FOUR

HOW ANSIBLE BUILDER EXECUTES

Ansible Builder can execute two separate steps:

- The first step is to create a build instruction file (`Containerfile` for Podman, `Dockerfile` for Docker) and a build context based on the execution environment definition file.
- The second step is to run a containerization tool (Podman or Docker) to build an image based on the build instruction file and build context.

The `ansible-builder build` command runs both steps.

The `ansible-builder create` command runs only the first step. For more details, read through the *CLI usage docs*.

4.1 How Ansible Builder builds images

Ansible Builder executes four stages when it runs your containerization tool to build a container image. The same four stages get executed if you build your container image directly with Podman or Docker, using a build instruction file and context generated by `ansible-builder create`. These stages are:

1. **Base**: uses Podman or Docker to pull the base image you defined, then installs the Python version (if defined and different from any Python on the base image), pip, ansible-runner, and ansible-core or ansible. All three later stages of the build process build on the output of the Base stage.
2. **Galaxy**: downloads the collections you defined from Galaxy and stashes them locally as files.
3. **Builder**: downloads the other packages (Python packages and system packages) you defined and stash them locally as files.
4. **Final**: integrates the first three stages, installing all the stashed files on the output of the Base stage and generating a new image that includes all the content.

Ansible Builder injects hooks at each stage of the container build process so you can add custom steps before and after every build stage.

You may need to install certain packages or utilities before the Galaxy and Builder stages. For example, if you need to install a collection from GitHub, you must install git after the Base stage to make it available during the Galaxy stage.

To add custom build steps, add an `additional_build_steps` section to your execution environment definition. For more details, read through the *CLI usage docs*.

CHAPTER

FIVE

DEFINING COLLECTION DEPENDENCIES

When Ansible Builder installs collections into an execution environment, it also installs each collection's dependencies if they are specified. Collection maintainers can learn to correctly declare dependencies for their collections from the *collection-level dependencies* page.

5.1 Installation

5.1.1 Requirements

- To build images, you must install a containerization tool - either `podman` or `docker` - as well as the `ansible-builder` Python package.
- The `--container-runtime` option must correspond to the containerization tool you use.
- `ansible-builder` version `3.x` requires Python `3.9` or higher.

5.1.2 Install from PyPI

```
$ pip3 install ansible-builder
```

Note: An **alternative** approach to installing `ansible-builder` is using the `ansible-dev-tools` package. Ansible Development Tools (ADT) is a single Python package that includes all necessary tools to set up a development environment, generate new collections, build and test the content.

```
# This also installs ansible-core if it is not already installed
$ pip3 install ansible-dev-tools
```

5.1.3 Install from Source

To install from the mainline development branch:

```
$ pip3 install https://github.com/ansible/ansible-builder/archive/devel.zip
```

To install from a specific tag or branch, replace <ref> in the following example:

```
$ pip3 install https://github.com/ansible/ansible-builder/archive/<ref>.zip
```

5.2 Execution environment definition

You define the content of your execution environment in a YAML file. By default, this file is called `execution-environment.yml` or `execution-environment.yaml`. This file tells Ansible Builder how to create the build instruction file (`Containerfile` for Podman, `Dockerfile` for Docker) and build context for your container image.

Note: This page documents the definition schema for Ansible Builder 3.x. If you are running an older version of Ansible Builder, you need an older schema version. Please consult older versions of the docs for more information. We recommend using version 3, which is more configurable and functional than previous versions.

5.2.1 Overview

The Ansible Builder 3.x execution environment definition file accepts seven top-level sections:

- additional_build_files
- additional_build_steps
- build_arg_defaults
- dependencies
- images

- options
- version

5.2.2 Version 3 sample files

Below are some sample version 3 EE files. To use Ansible Builder 3.x, you must specify the schema version. If your EE file does not specify `version: 3`, Ansible Builder will assume you want version 1.

This first simple example should successfully build an execution environment using the most recent `ubi9` base image and the most recently compatible versions of `ansible-core` and `ansible-runner` for that image.

```
---
version: 3
images:
  base_image:
    name: docker.io/redhat/ubi9:latest
dependencies:
  ansible_core:
    package_pip: ansible-core
  ansible_runner:
    package_pip: ansible-runner
```

The second example below is more detailed and is not usable without modification and/or the creation of supporting requirements and extra files, but does demonstrate more complete EE file syntax.

```
---
version: 3

build_arg_defaults:
  ANSIBLE_GALAXY_CLI_COLLECTION_OPTS: '--pre'

dependencies:
  ansible_core:
    package_pip: ansible-core==2.14.4
  ansible_runner:
    package_pip: ansible-runner
  galaxy: requirements.yml
  python:
    - six
    - psutil
  system: bindep.txt
  exclude:
    python:
      - docker
    system:
      - python3-Cython

images:
  base_image:
    name: docker.io/redhat/ubi9:latest
    # Other available base images:
    #   - quay.io/rockylinux/rockylinux:9
    #   - quay.io/centos/centos:stream9
```

(continues on next page)

(continued from previous page)

```
    #   - registry.fedoraproject.org/fedora:38
    #   - registry.redhat.io/ansible-automation-platform-23/ee-minimal-rhel8:latest
    #     (needs an account)

# Custom package manager path for the RHEL based images
# options:
#  package_manager_path: /usr/bin/microdnf

additional_build_files:
    - src: files/ansible.cfg
      dest: configs

additional_build_steps:
  prepend_base:
    - RUN echo This is a prepend base command!
    # Enable Non-default stream before packages provided by it can be installed.␣
↪(optional)
    # - RUN $PKGMGR module enable postgresql:15 -y
    # - RUN $PKGMGR install -y postgresql
  prepend_galaxy:
    - COPY _build/configs/ansible.cfg /etc/ansible/ansible.cfg

  prepend_final: |
    RUN whoami
    RUN cat /etc/os-release
  append_final:
    - RUN echo This is a post-install command!
    - RUN ls -la /etc
```

5.2.3 Configuration options

You may use the configuration YAML keys listed here in your v3 execution environment definition file.

additional_build_files

Specifies files to be added to the build context directory. These can then be referenced or copied by *additional_build_steps* during any build stage. The format is a list of dictionary values, each with a `src` and `dest` key and value.

Each list item must be a dictionary containing the following (non-optional) keys:

`src`
: Specifies the source file(s) to copy into the build context directory. This may either be an absolute path (e.g., `/home/user/.ansible.cfg`), or a path that is relative to the execution environment file. Relative paths may be a glob expression matching one or more files (e.g. `files/*.cfg`). Note that an absolute path may *not* include a regular expression. If `src` is a directory, the entire contents of that directory are copied to `dest`.

`dest`
: Specifies a subdirectory path underneath the `_build` subdirectory of the build context directory that should contain the source file(s) (e.g., `files/configs`). This may not be an absolute path or contain `..` within the path. This directory will be created for you if it does not exist.

additional_build_steps

Specifies custom build commands for any build phase. These commands will be inserted directly into the build instruction file for the container runtime (e.g., *Containerfile* or *Dockerfile*). The commands must conform to any rules required by the containerization tool.

You can add build steps before or after any stage of the image creation process. For example, if you need `git` to be installed before you install your dependencies, you can add a build step at the end of the `base` build stage.

Below are the valid keys for this section. Each supports either a multi-line string, or a list of strings.

prepend_base
: Commands to insert before building of the base image.

append_base
: Commands to insert after building of the base image.

prepend_galaxy
: Commands to insert before building of the galaxy image.

append_galaxy
: Commands to insert after building of the galaxy image.

prepend_builder
: Commands to insert before building of the builder image.

append_builder
: Commands to insert after building of the builder image.

prepend_final
: Commands to insert before building of the final image.

append_final
: Commands to insert after building of the final image.

Note: Please make sure that you do not specify *USER* directives in these build steps. This may lead to failures while building the image. If you want to override the *USER* setting, consider using the *options.user* setting mentioned below.

---

build_arg_defaults

Specifies default values for build args as a dictionary. This is an alternative to using the *--build-arg* CLI flag.

Build args used by `ansible-builder` are the following:

ANSIBLE_GALAXY_CLI_COLLECTION_OPTS
: This allows the user to pass the *–pre* flag (or others) to enable the installation of pre-release collections.

ANSIBLE_GALAXY_CLI_ROLE_OPTS
: This allows the user to pass any flags, such as *–no-deps*, to the role installation.

PKGMGR_PRESERVE_CACHE
: This controls how often the package manager cache is cleared during the image build process. If this value is not set, which is the default, the cache is cleared frequently. If it is set to the string *always*, the cache is never cleared. Any other value forces the cache to be cleared only after the system dependencies are installed in the final build stage.

Ansible Builder hard-codes values given inside of `build_arg_defaults` into the build instruction file, so they will persist if you run your container build manually.

If you specify the same variable in the execution environment definition and at the command line with the CLI *--build-arg* flag, the CLI value will take higher precedence (the CLI value will override the value in the execution environment definition).

dependencies

Specifies dependencies to install into the final image, including `ansible-core`, `ansible-runner`, Python packages, system packages, and Ansible Collections. Ansible Builder automatically installs dependencies for any Ansible Collections you install.

In general, you can use standard syntax to constrain package versions. Use the same syntax you would pass to `dnf`, `pip`, `ansible-galaxy`, or any other package management utility. You can also define your packages or collections in separate files and reference those files in the `dependencies` section of your execution environment definition file.

The following keys are valid for this section:

ansible_core
: The version of the `ansible-core` Python package to be installed. This value is a dictionary with a single key, `package_pip`. The `package_pip` value is passed directly to *pip* for installation and can be in any format that pip supports. Below are some example values:

```
ansible_core:
    package_pip: ansible-core
ansible_core:
    package_pip: ansible-core==2.14.3
ansible_core:
    package_pip: https://github.com/example_user/ansible/archive/refs/
↪heads/ansible.tar.gz
```

ansible_runner
: The version of the Ansible Runner Python package to be installed. This value is a dictionary with a single key, `package_pip`. The `package_pip` value is passed directly to *pip* for installation and can be in any format that pip supports. Below are some example values:

```
ansible_runner:
    package_pip: ansible-runner
ansible_runner:
    package_pip: ansible-runner==2.3.2
ansible_runner:
    package_pip: https://github.com/example_user/ansible-runner/archive/
↪refs/heads/ansible-runner.tar.gz
```

galaxy
: Ansible Collections to be installed from Galaxy. This may be a filename, a dictionary, or a multi-line string representation of an Ansible Galaxy `requirements.yml` file (see below for examples). Read more about the requirements file format in the Galaxy user guide.

python
: The Python installation requirements. This may either be a filename, or a list of requirements (see below for an example).

Note: Python requirement specifications are expected to be limited to features defined by PEP 508. Hash tag comments will always be allowed. Any deviation from this specification will be

passed through to pip unverified and unaltered, although this is considered undefined and unsupported behavior. It is not recommended that you depend on this behavior.

---

python_interpreter
A dictionary that defines the Python system package name to be installed by `dnf` (`package_system`) and/or a path to the Python interpreter to be used (`python_path`).

system
The system packages to be installed, in bindep format. This may either be a filename, or a list of requirements (see below for an example).

exclude
A dictionary defining the Python or system requirements to be excluded from the top-level dependency requirements of referenced collections. These exclusions will not apply to the user supplied Python or system dependencies, nor will they apply to dependencies of dependencies (top-level only).

The following keys are valid for this section:

- `python` - A list of Python dependencies to be excluded.
- `system` - A list of system dependencies to be excluded.
- `all_from_collections` - If you want to exclude *all* Python and system dependencies from one or more collections, supply a list of collection names under this key.

The exclusion feature supports two forms of matching:

- Simple name matching.
- Advanced name matching using regular expressions.

For simple name matching, you need only supply the name of the requirement/collection to match. All values will be compared in a case-insensitive manner.

For advanced name matching, begin the exclusion string with the tilde (~) character to indicate that the remaining portion of the string is a regular expression to be used to match a requirement/collection name. The regex should be considered case-insensitive.

Note: The regular expression must match the full requirement/collection name. For example, `~foo.` does not fully match the name `foobar`, but `~foo.+` does.

With both forms of matching, the exclusion string will be compared against the *simple* name of any Python or system requirement. For example, if you need to exclude the system requirement that appears as `foo [!platform:gentoo]` within an included collection, then your exclusion string should be `foo`. To exclude the Python requirement `bar == 1.0.0`, your exclusion string would be `bar`.

Example using both simple and advanced matching:

```
dependencies:
    exclude:
        python:
            - docker
        system:
            - python3-Cython
        all_from_collections:
            # Regular expression to exclude all from community collections
            - ~community\..+
```

Note: The `exclude` option requires `ansible-builder` version `3.1` or newer.

The following example uses filenames that contain various dependencies:

```
dependencies:
    python: requirements.txt
    system: bindep.txt
    galaxy: requirements.yml
    ansible_core:
        package_pip: ansible-core==2.14.2
    ansible_runner:
        package_pip: ansible-runner==2.3.1
    python_interpreter:
        package_system: "python310"
        python_path: "/usr/bin/python3.10"
```

And this example uses inline values:

```
dependencies:
    python:
      - pywinrm
    system:
      - iputils [platform:rpm]
    galaxy:
      collections:
        - name: community.windows
        - name: ansible.utils
          version: 2.10.1
    ansible_core:
        package_pip: ansible-core==2.14.2
    ansible_runner:
        package_pip: ansible-runner==2.3.1
    python_interpreter:
        package_system: "python310"
        python_path: "/usr/bin/python3.10"
```

images

Specifies the base image to be used. At a minimum you *MUST* specify a source, image, and tag for the base image. The base image provides the operating system and may also provide some packages. We recommend using the standard `host/namespace/container:tag` syntax to specify images. You may use Podman or Docker shortcut syntax instead, but the full definition is more reliable and portable.

Valid keys for this section are:

base_image
: A dictionary defining the parent image for the execution environment. A `name` key must be supplied with the container image to use. Use the `signature_original_name` key if the image is mirrored within your repository, but signed with the original image's signature key.

image verification

You can verify signed container images if you are using the `podman` container runtime. Set the *--container-policy* CLI option to control how this data is used with a Podman policy.json file for container image signature validation.

- `ignore_all` policy: Generate a *policy.json* file in the build *context directory* where no signature validation is performed.
- `system` policy: Signature validation is performed using pre-existing *policy.json* files in standard system locations. `ansible-builder` assumes no responsibility for the content within these files, and the user has complete control over the content.
- `signature_required` policy: `ansible-builder` will use the container image definitions here to generate a *policy.json* file in the build *context directory* that will be used during the build to validate the images.

options

A dictionary of keywords/options that can affect builder runtime functionality. Valid keys for this section are:

container_init
: A dictionary with keys that allow for customization of the container `ENTRYPOINT` and `CMD` directives (and related behaviors). Customizing these behaviors is an advanced task, and may result in subtle, difficult-to-debug failures. As the provided defaults for this section control several intertwined behaviors, overriding any value will skip all remaining defaults in this dictionary. Valid keys are:

  cmd
  : Literal value for the `CMD` Containerfile directive. The default value is `["bash"]`.

  entrypoint
  : Literal value for the `ENTRYPOINT` Containerfile directive. The default entrypoint behavior handles signal propagation to subprocesses, as well as attempting to ensure at runtime that the container user has a proper environment with a valid writeable home directory, represented in `/etc/passwd`, with the `HOME` envvar set to match. The default entrypoint script may emit warnings to `stderr` in cases where it is unable to suitably adjust the user runtime environment. This behavior can be ignored or elevated to a fatal error; consult the source for the `entrypoint` target script for more details. The default value is `["/opt/builder/bin/entrypoint", "dumb-init"]`.

  package_pip
  : Package to install via pip for entrypoint support. This package will be installed in the final build image. The default value is `dumb-init==1.2.5`.

package_manager_path
: A string with the path to the package manager (For example - `dnf` or `microdnf`) to use. The default is `/usr/bin/dnf`. This value will be used to install a Python interpreter, if specified in `dependencies`, and during the build phase by the `assemble` script.

skip_ansible_check
: This boolean value controls whether or not the check for an installation of Ansible and Ansible Runner is performed on the final image. Set this value to `True` to not perform this check. The default is `False`.

skip_pip_install
: This boolean value controls whether or not we attempt to install pip into the base image. Pip is necessary for Python requirement installation, among other things. You may choose to disable this step and handle installing pip manually if the current method of pip installation does not work for you. The default is `False`.

relax_passwd_permissions
: This boolean value controls whether the `root` group (GID 0) is explicitly granted write permission to `/etc/passwd` in the final container image. The default entrypoint script may attempt to update `/etc/passwd` under some container runtimes with dynamically created users to ensure a fully functional POSIX user environment and home directory. Disabling this capability can cause failures of software features that require users to be listed in `/etc/passwd` with a valid and writeable home directory (eg, `async` in ansible-core, and the `~username` shell expansion). The default is `True`.

workdir
: Default current working directory for new processes started under the final container image. Some container runtimes also use this value as `HOME` for dynamically-created users in the `root` (GID 0) group. When this value is specified, the directory will be created (if it doesn't already exist), set to `root` group ownership, and `rwx` group permissions recursively applied to it. The default value is `/runner`.

user
: This sets the username or UID to use as the default user for the final container image. The default value `1000`.

tags
: Specifies the names that are assigned to the resulting image if the build process completes successfully. The default value is `ansible-execution-env:latest`.

Example `options` section:

```
options:
    container_init:
        package_pip: dumb-init>=1.2.5
        entrypoint: '["dumb-init"]'
        cmd: '["csh"]'
    package_manager_path: /usr/bin/microdnf
    relax_passwd_permissions: false
    skip_ansible_check: true
    workdir: /myworkdir
    user: bob
    tags:
      - ee_development:latest
```

version

An integer value that sets the schema version of the execution environment definition file. Defaults to `1`. Must be `3` if you are using Ansible Builder 3.x.

5.3 CLI Usage

Ansible Builder can execute two separate steps. The first step is to create a build instruction file (Containerfile for Podman, Dockerfile for Docker) and a build context based on your *definition* file. The second step is to run a containerization tool (Podman or Docker) to build an image based on the build instruction file and build context. The `ansible-builder build` command executes both steps, giving you a build instruction file, a build context, and a fully built container image. The `ansible-builder create` command only executes the first step, giving you a build instruction file and a build context. If you use `ansible-builder create`, you can use the resulting build instruction file and build context to build your container images on the platform of your choice.

5.3.1 The `build` command

The `ansible-builder build` command:

- takes an *execution environment definition file* as an input,
- outputs a build instruction file (Containerfile for Podman, Dockerfile for Docker),
- creates a build context necessary for building an execution environment image,
- builds the image.

By default, it looks for a file named `execution-environment.yml` (or `execution-environment.yaml`) in the current directory.

To build an execution environment using the default definition file, run:

```
$ ansible-builder build
Running command:
  podman build -f context/Containerfile -t ansible-execution-env:latest context
Complete! The build context can be found at: /path/to/context
```

Ansible Builder produces a ready-to-use container image and preserves the build context, which you can use to rebuild the image at a different time and/or location with the tooling of your choice.

5.3.2 Flags for the `build` command

`--tag`

Customizes the tagged name applied to the built image. To create an image with a custom name:

```
$ ansible-builder build --tag=my-custom-ee
```

More recent versions of `ansible-builder` support multiple tags:

```
$ ansible-builder build --tag=tag1 --tag=tag2
```

`--file`

Specifies the execution environment file. To use a file other than the default:

```
$ ansible-builder build --file=my-ee-def.yml
```

`--galaxy-keyring`

Specifies a keyring for `ansible-galaxy` to use to verify collection signatures during installation. To verify collection signatures:

```
$ ansible-builder create --galaxy-keyring=/path/to/pubring.kbx
$ ansible-builder build --galaxy-keyring=/path/to/pubring.kbx
```

If you do not pass this option, no signature verification is performed. If you do pass this option, but the version of Ansible is too old to support this feature, you will see an error during the image build process.

`--galaxy-ignore-signature-status-code`

Ignores certain errors that may occur while verifying collections. This option is passed unmodified to `ansible-galaxy` calls. Valid only when `--galaxy-keyring` is also set. See the `ansible-galaxy` documentation for more information.

```
$ ansible-builder create --galaxy-keyring=/path/to/pubring.kbx --galaxy-ignore-signature-
↪status-code 500
$ ansible-builder build --galaxy-keyring=/path/to/pubring.kbx --galaxy-ignore-signature-
↪status-code 500
```

`--galaxy-required-valid-signature-count`

Overrides the number of required valid collection signatures. This option is passed unmodified to `ansible-galaxy` calls. Valid only when `--galaxy-keyring` is also set. See the `ansible-galaxy` documentation for more information.

```
$ ansible-builder create --galaxy-keyring=/path/to/pubring.kbx --galaxy-required-valid-
↪signature-count 3
$ ansible-builder build --galaxy-keyring=/path/to/pubring.kbx --galaxy-required-valid-
↪signature-count 3
```

--context

Specifies the directory name for the build context Ansible Builder creates. Default directory name is `context` in the current working directory. To specify another location:

```
$ ansible-builder build --context=/path/to/dir
```

--build-arg

Passes build-time arguments to Podman or Docker. Specify these flags or variables the same way you would with `podman build` or `docker build`.

By default, the Containerfile / Dockerfile created by Ansible Builder contains a build argument `EE_BASE_IMAGE`, which can be useful for rebuilding execution environments without modifying any files.

```
$ ansible-builder build --build-arg FOO=bar
```

To use different build arguments, you can specify `--build-arg` multiple times:

```
$ ansible-builder build --build-arg FOO=bar --build-arg SIMPLE=sample
```

To use a custom base image:

```
$ ansible-builder build --build-arg EE_BASE_IMAGE=registry.example.com/another-ee
```

--container-runtime

Specifies the containerization tool used to build images. Default is Podman. To use Docker:

```
$ ansible-builder build --container-runtime=docker
```

--container-policy

Note: Added in version 1.2

Specifies the container image validation policy to use. Valid only when *--container-runtime* is `podman`. Valid values are one of:

- `ignore_all`: Run podman with generated policy that ignores all signatures.
- `system`: Relies on podman's consumption of system policy/signature with inline keyring paths. No builder-specific overrides are possible.
- **`signature_required`: Run podman with `--pull-always` and a generated** policy that rejects all by default, with generated identity requirements for referenced container images, using an explicitly-provided keyring (specified with the *--container-keyring* CLI option).

--container-keyring

Note: Added in version 1.2

Specifies the path to a GPG keyring file to use for validating container image signatures.

--extra-build-cli-args

Note: Added in version 3.1

This option allows the user to pass any additional command line arguments to the container engine build command (`docker build` or `podman build`). Take care when using this option as there is no attempt to identify or resolve conflicting argument values from this option and arguments normally added by `ansible-builder`.

```
$ ansible-builder build --extra-build-cli-args='--pull --env=MY_ENV_VAR'
```

--verbosity

Customizes the level of verbosity:

```
$ ansible-builder build --verbosity 2
```

You may also use `-v` for the shorthand version. You may either specify an integer for the verbosity level, or supply multiples of the option. Individual instances of `-v` will stack. For example, the following are equivalent to setting the verbosity level to `3`:

```
$ ansible-builder build -v 3
$ ansible-builder build -vvv
$ ansible-builder build -v -v -v
```

--prune-images

Removes unused images created after the build process:

```
$ ansible-builder build --prune-images
```

Note: This flag removes all the dangling images on the given machine whether they already existed or were created by `ansible-builder` build process.

`--squash`

Controls the final image layer squashing. Valid values are:

- `new`: Squash all of the final image's new layers into a single new layer (preexisting layers are not squashed).
- `all`: Squash all of the final image's layers, including those inherited from the base image, into a single new layer.
- `off`: Turn off layer squashing. This is the default.

Note: This flag is compatible only with the `podman` runtime and will be ignored for any other runtime. Docker does not support layer squashing; it is considered an experimental feature.

5.3.3 The `create` command

The `ansible-builder create` command accepts an execution environment definition as an input and outputs the build context necessary for building an execution environment image. However, the `create` command *will not* build the execution environment image; this is useful for creating just the build context and a `Containerfile` that can then be shared.

5.3.4 Examples

The example in `test/data/pytz` requires the `awx.awx` collection in the execution environment definition. The lookup plugin `awx.awx.schedule_rrule` requires the PyPI `pytz` and another library to work. If `test/data/pytz/execution-environment.yml` file is given to the `ansible-builder build` command, then it will install the collection inside the image, read `requirements.txt` inside of the collection, and then install `pytz` into the image.

The image produced can be used inside of an `ansible-runner` project by placing these variables inside the `env/settings` file, inside of the private data directory.

```
---
container_image: image-name
process_isolation_executable: podman # or docker
process_isolation: true
```

The `awx.awx` collection is a subset of content included in the default AWX execution environment. More details can be found at the awx-ee repository.

5.3.5 Deprecated Features

The `--base-image` CLI option has been removed. See the `--build-arg` option for a replacement.

5.4 Collection-level dependencies

When Ansible Builder installs collections into an execution environment, it also installs their controller-side Python or system package dependencies listed by each collection on Galaxy.

For Ansible Builder to find and install collection dependencies, those dependencies must be defined in files in a collection repository.

Note: If present, the files below must be included in the packaged collection on Galaxy. Ansible Builder cannot install dependencies listed in files that are included in the `build_ignore` of a collection, because those files are not included in the collection artifact.

If you are a collection maintainer, make sure the controller-side dependencies are specified and *verified*.

We recommend you specify paths to dependency files in the `meta/execution-environment.yml` file. Here is an example of its content:

```
dependencies:
  python: meta/ee-requirements.txt  # List Python package requirements in the file
  system: meta/ee-bindep.txt  # List system package requirements in the file
```

If the `meta/execution-environment.yml` file is not present, by default, Ansible Builder will expect the dependencies to be defined in:

- the `requirements.txt` file in the collection root directory for Python package requirements
- the `bindep.txt` file in the collection root directory for system package requirements

---

Note: If your collection uses the `requirements.txt` or `bindep.txt` files in its root directory for anything else but its controller-side dependencies, for example, for listing testing requirements, make sure you use the `meta/execution-environment.yml` file to specify other dependency files for execution environment purposes.

5.5 Dependency introspection

If any dependencies are given, the introspection is run by Ansible Builder so that the requirements are found before container image assembly.

A user can see the introspection output during the builder intermediate phase using the `build -v3` option.

5.6 How to verify collection-level metadata

Note: Running the introspect command described below is not part of a typical workflow for building and using execution environments.

Collection developers can verify that dependencies specified in the collection will be processed correctly by Ansible Builder.

To do that, the collection has to be installed locally.

5.6.1 When installing collections using ansible-galaxy

The easiest way to install a collection is to use the ansible-galaxy command which is a part of the `ansible` package.

Run the `introspect` command against your collection path:

```
ansible-builder introspect COLLECTION_PATH
```

The default collection path used by the `ansible-galaxy` command is `~/.ansible/collections/`. Read more about collection paths in the Ansible configuration settings guide.

Note: Use the `-v3` option to `introspect` to see logging messages about requirements that are being excluded.

5.6.2 When installing collections manually

If you download collection tarballs from Galaxy manually or clone collection git repositories, for the `introspect` command to work properly, be sure you store your collections using the following directory structure:

```
ansible_collections/NAMESPACE/COLLECTION
```

For example, if you need to inspect the `community.docker` collection, the path will be:

```
ansible_collections/community/docker
```

Then, if the `ansible_collection` directory is in your home directory, you can run `introspect` with the following command:

```
ansible-builder introspect ~/
```

Python Dependencies

Ansible Builder combines all the Python requirements files from all collections into a single file.

Certain package names are specifically *ignored* by `ansible-builder`, meaning that Ansible Builder does not include them in the combined file of Python dependencies, even if a collection lists them as dependencies. These include test packages and packages that provide Ansible itself. The full list can be found in `EXCLUDE_REQUIREMENTS` in `src/ansible_builder/_target_scripts/introspect.py`.

If you need to include one of these ignored package names, use the `--user-pip` option of the `introspect` command to list it in the user requirements file. Packages supplied this way are not processed against the list of excluded Python packages.

Note: These dependencies are subject to the same PEP 508 format restrictions described for Python requirements in the *EE definition specification*.

System-level Dependencies

For system packages, use the `bindep` format to specify cross-platform requirements, so they can be installed by whichever package management system the execution environment uses. Collections should specify necessary requirements for `[platform:rpm]`.

Ansible Builder combines system package entries from multiple collections into a single file.

- Requirements with `compile` profile indicate that these requirements are needed to install other requirements (especially Python ones), but are not required to be in the final build.
- Requirements with `epel` profile indicate that EPEL repositories will be enabled before installing these requirements.
- Only requirements with *no* profiles (runtime requirements) are installed to the image.

Entries from multiple collections which are outright duplicates of each other may be consolidated in the combined file.

5.7 Ansible Builder Porting Guides

This section lists porting guides that can help you in updating your Execution Environment files between versions of `ansible-builder`.

5.7.1 Ansible Builder 3.1 Porting Guide

This section discusses the behavioral changes between `ansible-builder` version 3.0 and version 3.1.

Note: We highly advise running `ansible-builder` with increased verbosity using the `-vvv` option (`--v3` for versions older than 3.1) to fully expose any error messages that may help in diagnosing any problems.

Topics

Python Requirements Handling

The 3.1 release significantly changes how Python and system requirements are handled by simplifying dependency parsing. This release removes the use of an external library that was unmaintained and parsed many types of Python dependencies either partially or completely incorrectly. The changes are described below.

PEP 508 Standard

Python requirements files are expected to follow the PEP 508 standard. Builder will *expect* the requirements file to be in this format, but it makes two exceptions:

1. Comments (lines beginning with `#`) are ignored.
2. Any line from the requirements file that is not compliant with PEP508 causes a warning to be emitted and the line passed through to `pip` unmodified. It is not recommended to depend on this behavior, as it can change suddenly between `pip` releases, and can cause other problems with dependency resolution.

The passthrough of non-PEP508 compliant lines may expose issues that were hidden by the version 3.0 dependency sanitizer, which often silently ignored and removed them.

Dependency Sanitization

Dependency sanitization (the combining of duplicate dependencies into a single dependency entry) is no longer performed by `ansible-builder`.

Note: The `--sanitize` option to the `ansible-builder introspect` command still exists, but is now undocumented and does nothing.

The effect of this change is that builder will now pass a listed dependency multiple times for each requirement file in which it is found. For example, with version 3.0, if collection A listed the Python dependency `foo`, and collection B listed the dependency `foo>=1.0`, then it would have appeared in the combined Python requirements file as a single entry:

```
foo,foo>=1.0   # from collection A, B
```

Now, with version 3.1, those dependencies are no longer combined and will appear in the combined Python requirements file as separate entries:

```
foo  # from collection A
foo>=1.0  # from collection B
```

If your container image has an older version of `pip`, this change might cause an error during the image build process. This guide covers how to deal with this situation below.

Common Issues

This section lists some common errors that might be encountered when running `ansible-builder` version 3.1.

ERROR: Double requirement given

Because dependency sanitization has been removed, *all* Python requirements from included collections and user requirement files are passed along to the `pip` command that installs those requirements. Due to this change, the image build may exit abnormally with an error similar to the following:

```
ERROR: Double requirement given: netaddr>=0.10.1 (from -r /tmp/src/requirements.txt␣
↪(line 13)) (already in netaddr (from -r /tmp/src/requirements.txt (line 4)), name=
↪'netaddr')
Error: building at STEP "RUN /output/scripts/assemble": while running runtime: exit␣
↪status 1
```

This error comes from `pip` within one of the intermediate container images when attempting to install the Python requirements from included collections and/or from the user supplied requirements. This intermediate image is based on the base image defined within the Execution Environment file, and it means that the version of `pip` installed within that image is too old to handle duplicate requirement entries.

To determine where the duplicate requirements are coming from, run `ansible-builder` with the `-vvv` option to get more verbose output, then look for the output from the introspection phase. It will look similar to:

```
[3/4] STEP 12/13: RUN $PYCMD /output/scripts/introspect.py introspect --write-bindep=/
↪tmp/src/bindep.txt --write-pip=/tmp/src/requirements.txt
Creating parent directory for /tmp/src/requirements.txt
---
python:
- 'netaddr  # from collection ansible.netcommon'
- 'netaddr>=0.10.1  # from collection ansible.utils'
```

In the example output above, the double Python requirement for `netaddr` is coming from the collections `ansible.netcommon` and `ansible.utils`.

The solution requires upgrading `pip` within the base image to a version that contains an updated dependency resolver. Beginning with `pip` version 20.3, the dependency resolver can handle duplicate requirements whose versions do not conflict, so that version is the minimum required. Upgrade `pip` in the base image from within the Execution Environment file by adding these lines to it:

```
additional_build_steps:
  append_base:
    - RUN $PYCMD -m pip install -U pip
```

That will upgrade `pip` to the latest version within the base image. To restrict the upgrade to a specific version of `pip`, alter the upgrade command to specify that version. For example:

```
additional_build_steps:
  append_base:
    - RUN $PYCMD -m pip install -U pip==20.3
```

5.7.2 Ansible Builder 3.0 Porting Guide

This section discusses the behavioral changes between `ansible-builder` version 1.2 and version 3.0.

We suggest you read this page along with ansible-builder 3.0 release notes to understand what updates you may need to make.

Topics

Overview

The Ansible Builder 3.x execution environment definition file accepts seven top-level sections:

- additional_build_files
- additional_build_steps
- build_arg_defaults
- dependencies
- images
- options
- version

Porting options

You need to change the configuration YAML keys in your older (v1 and v2) execution environment file to the one listed here.

additional_build_files

This is a new configuration that can be used to specify files to be added to the build context directory. These can then be referenced or copied by *additional_build_steps* during any build stage.

See the *additional_build_files* section for more details.

additional_build_steps

With Ansible Builder 3, you can specify more fine-grained build steps or custom commands for any build phase. These commands will be inserted directly into the build instruction file for the container runtime (For example, *Containerfile* or *Dockerfile*). The commands must conform to any rules required by the containerization tool.

These are additional build steps -

- `prepend_base`
- `append_base`
- `prepend_galaxy`
- `append_galaxy`
- `prepend_builder`
- `append_builder`
- `prepend_final`
- `append_final`

See the *additional_build_steps* section for more details.

dependencies

Specifies dependencies to install into the final image, including `ansible-core`, `ansible-runner`, Python packages, system packages, and Ansible Collections. Ansible Builder automatically installs dependencies for any Ansible Collections you install.

In general, you can use standard syntax to constrain package versions. Use the same syntax you would pass to `dnf`, `pip`, `ansible-galaxy`, or any other package management utility. You can also define your packages or collections in separate files and reference those files in the `dependencies` section of your execution environment definition file.

The following keys are valid for this section:

- `ansible_core`
- `ansible_runner`
- `galaxy`
- `python`
- `python_interpreter`
- `system`

See the *dependencies* section for more details.

images

Specifies the base image to be used. At a minimum you *MUST* specify a source, image, and tag for the base image. The base image provides the operating system and may also provide some packages. We recommend using the standard `host/namespace/container:tag` syntax to specify images.

See the *images* section for more details.

image verification

You can verify signed container images if you are using the `podman` container runtime.

See the *image verification* section for more details.

options

A dictionary of keywords/options that can affect builder runtime functionality. Valid keys for this section are:

- `container_init`
- `cmd`
- `entrypoint`
- `package_pip`
- `package_manager_path`
- `skip_ansible_check`
- `relax_passwd_permissions`
- `workdir`
- `user`
- `tags`

See the *options* section for more details.

version

Must be `3` if you are using Ansible Builder 3.x.

See the *version* section for more details.

5.8 Glossary

execution environment
: execution environments are container images intended to be used as Ansible control nodes. Starting in version 2.0, ansible-runner. can make use of these images.

Control node
: The machine or container running Ansible. See the Ansible documentation for a more comprehensive explanation.

5.9 Copying arbitrary files to EE

Ansible Builder version 3 schema provides the option to copy files to an EE image. See the *version 3 schema* for more details.

In the example below, we will take a look at copying arbitrary files to an execution environment.

```
---
version: 3

images:
  base_image:
    name: quay.io/centos/centos:stream9  # vanilla image

dependencies:
  # Use Python 3.9
  python_interpreter:
    package_system: python39
    python_path: /usr/bin/python3.9
  # Collections to be installed
  galaxy:
    collections:
      - ansible.utils

additional_build_files:
  # copy arbitrary files next to this EE def into the build context - we can refer to␣
↪them later...
  - src: files/rootCA.crt
    dest: configs

additional_build_steps:
  prepend_base:
    # copy a custom CA cert into the base image and recompute the trust database
    # because this is in "base", all stages will inherit (including the final EE)
    - COPY _build/configs/rootCA.crt /usr/share/pki/ca-trust-source/anchors
    - RUN update-ca-trust
```

In this example, the *additional_build_files* section allows you to add *rootCA.crt* to the build context directory. Once this file is copied to the build context directory, it can be used in the build process. In order to use, the file, we need to copy it from the build context directory using the *COPY* directive specified in the *prepend_base* step of *additional_build_steps* section.

Finally, you can perform any action based upon the copied file, such as in this example updating dynamic configuration of CA certificates by running *RUN update-ca-trust*.

See also:

Execution Environment Definition version 3
: The detailed documentation about EE definition version 3

5.10 Building EEs with environment variables

Ansible Builder version 3 schema provides the option to specify environment variables that can be used in the build process. See *version 3 schema* for more details.

In the example below, we will take a look at specifying *ENV* variables.

```
---
version: 3

images:
  base_image:
    name: quay.io/centos/centos:stream9

dependencies:
  python_interpreter:
    package_system: python39
    python_path: /usr/bin/python3.9
  ansible_core:
    package_pip: ansible-core==2.14.0
  ansible_runner:
    package_pip: ansible-runner==2.3.2

additional_build_steps:
  prepend_base:
    - ENV FOO=bar
    - RUN echo $FOO > /tmp/file1.txt
```

In this example, we are specifying an environment variable that may be required for the build process. In order to achieve this functionality we are using the *ENV* variable definition in the `prepend_base` step of the *additional_build_steps* section.

We can use the same environment variable in the later stage of the build process.

See also:

***Execution Environment Definition version 3*.**
The detailed documentation about EE definition version 3

5.11 Building EEs with environment variables for Galaxy configuration

Ansible Builder version 3 schema allows users to perform complex scenarios such as specifying custom Galaxy configurations. You can use this approach to pass sensitive information, such as authentication tokens, into the EE build without leaking them into the final EE image.

In the example below, we will take a look at

- Using Galaxy Server environment variables

```
---
version: 3

images:
```

(continues on next page)

(continued from previous page)

```
  base_image:
    # Needs login
    name: registry.redhat.io/ansible-automation-platform-23/ee-minimal-rhel8:latest

dependencies:
  # No need to specify ansible-core or ansible-runner dependencies
  # because they are included in the base image.

  # Collections to be installed using Galaxy
  galaxy:
    collections:
      - ansible.utils

additional_build_steps:
  prepend_galaxy:
    # Environment variables used for Galaxy client configurations
    - ENV ANSIBLE_GALAXY_SERVER_LIST=automation_hub
    - ENV ANSIBLE_GALAXY_SERVER_AUTOMATION_HUB_URL=https://console.redhat.com/api/
↪automation-hub/content/xxxxxxx-synclist/
    - ENV ANSIBLE_GALAXY_SERVER_AUTOMATION_HUB_AUTH_URL=https://sso.redhat.com/auth/
↪realms/redhat-external/protocol/openid-connect/token
    # define a custom build arg env passthru - we still also have to pass
    # `--build-arg ANSIBLE_GALAXY_SERVER_AUTOMATION_HUB_TOKEN` to get it to pick it up␣
↪from the env
    - ARG ANSIBLE_GALAXY_SERVER_AUTOMATION_HUB_TOKEN

options:
  package_manager_path: /usr/bin/microdnf  # downstream images use non-standard package␣
↪manager
```

You can provide environment variables such as `ANSIBLE_GALAXY_SERVER_LIST`, `ANSIBLE_GALAXY_SERVER_AUTOMATION_HUB_URL` and `ANSIBLE_GALAXY_SERVER_AUTOMATION_HUB_AUTH_URL` using the `ENV` directive. See configuring Galaxy client for more details.

For security reasons, we do not want to store sensitive information in this case *ANSIBLE_GALAXY_SERVER_AUTOMATION_HUB_TOKEN*. You can use *ARG* directive to receive sensitive information from the user as input. *–build-args* can be used to provide this information while invoking the *ansible-builder* command.

See also:

Execution Environment Definition version 3
The detailed documentation about EE definition version 3

5.12 Passing Secrets

When creating an Execution Environment, it may be useful to use build secrets. This can be done with a combination of the use of *additional_build_steps* within the EE definition file, and the *--extra-build-cli-args* CLI option.

Use the *--extra-build-cli-args* CLI option to pass a build CLI argument that defines the secret:

```
ansible-builder build --extra-build-cli-args="--secret id=mytoken,src=my_secret_file.txt"
```

Then, use a custom `RUN` command within your EE definition file that references this secret:

```
---
version: 3

images:
  base_image:
    name: quay.io/centos/centos:stream9

additional_build_steps:
  prepend_base:
    - RUN --mount=type=secret,id=mytoken TOKEN=$(cat /run/secrets/mytoken) some_command

options:
  skip_ansible_check: true
```

5.13 Validating Installed Python Dependencies

It is a good idea to verify that all of the installed Python packages have compatible dependencies installed when the final container image is built. This is especially important if you have excluded any external collection dependencies and manually specified any replacements.

The `pip` utility includes a check option that can perform this validation. When `pip check` is run, it will do the validation of the currently installed Python packages, and if any errors are identified, it will exit with a non-zero status. A good place to call this check is during the end of the final build phase, just after all dependencies have been installed. Add the code below to your execution environment file to do the validation, which will fail the build if Python dependencies are not satisfied:

```
additional_build_steps:
  append_final:
    - RUN $PYCMD -m pip check
```

Using the `$PYCMD -m pip` calling form, instead of calling `pip` directly, will guarantee that the same Python executable that was used to install the Python packages is used to do the validation.

INDEX