Creating and Consuming Execution Environments

Create and use execution environments with Ansible Builder
Create and use execution environments with Ansible Builder
Abstract

This guide shows how to create consistent and reproducible automation execution environments for your Red Hat Ansible Automation Platform.
# Table of Contents

**PREFACE** ................................................................. 4

**MAKING OPEN SOURCE MORE INCLUSIVE** ........................................ 5

**PROVIDING FEEDBACK ON RED HAT DOCUMENTATION** .......................... 6

**CHAPTER 1. INTRODUCTION TO AUTOMATION EXECUTION ENVIRONMENTS** ............ 7

1.1. ABOUT AUTOMATION EXECUTION ENVIRONMENTS .............................. 7

1.1.1. Why use automation execution environments? ............................... 7

**CHAPTER 2. USING ANSIBLE BUILDER** ........................................ 8

2.1. WHY USE ANSIBLE BUILDER? .................................................. 8

2.2. INSTALLING ANSIBLE BUILDER ................................................ 8

2.3. BUILDING A DEFINITION FILE ................................................. 8

2.4. BUILDING THE AUTOMATION EXECUTION ENVIRONMENT IMAGE .......... 10

2.5. BREAKDOWN OF DEFINITION FILE CONTENT .................................. 10

2.5.1. Build args and base image ................................................... 10

2.5.1.1. Galaxy .............................................................................. 11

2.5.1.2. Python .............................................................................. 11

2.5.1.3. System .............................................................................. 12

2.5.2. Images .................................................................................. 12

2.5.3. Additional build files ................................................................ 13

2.5.4. Additional custom build steps .................................................. 13

2.5.5. Additional resources ............................................................... 14

2.6. OPTIONAL BUILD COMMAND ARGUMENTS ................................... 14

2.7. CONTAINERFILE ....................................................................... 15

2.8. CREATING A CONTAINERFILE WITHOUT BUILDING AN IMAGE ..... 15

**CHAPTER 3. COMMON AUTOMATION EXECUTION ENVIRONMENT SCENARIOS** ............. 16

3.1. UPDATING THE AUTOMATION HUB CA CERTIFICATE ....................... 16

3.2. USING AUTOMATION HUB AUTHENTICATION DETAILS WHEN BUILDING AUTOMATION EXECUTION ENVIRONMENTS ............................................. 16

3.3. ADDITIONAL RESOURCES ....................................................... 17

**CHAPTER 4. PUBLISHING AN AUTOMATION EXECUTION ENVIRONMENT** ............... 18

4.1. CUSTOMIZING AN EXISTING AUTOMATION EXECUTION ENVIRONMENTS IMAGE ............................................................... 18

4.2. ADDITIONAL RESOURCES (OR NEXT STEPS) .................................. 20

**CHAPTER 5. POPULATING YOUR PRIVATE AUTOMATION HUB CONTAINER REGISTRY** ............. 21

5.1. PULLING IMAGES FOR USE IN AUTOMATION HUB ............................ 21

5.2. TAGGING IMAGES FOR USE IN AUTOMATION HUB ........................... 22

5.3. PUSHING A CONTAINER IMAGE TO PRIVATE AUTOMATION HUB ........ 22

**CHAPTER 6. SETTING UP YOUR CONTAINER REPOSITORY** ............................ 24

6.1. PREREQUISITES TO SETTING UP YOUR CONTAINER REGISTRY ........... 24

6.2. ADDING A README TO YOUR CONTAINER REPOSITORY .................. 24

6.3. PROVIDING ACCESS TO YOUR CONTAINER REPOSITORY ................. 24

6.4. TAGGING CONTAINER IMAGES .................................................. 25

6.5. CREATING A CREDENTIAL IN AUTOMATION CONTROLLER ................ 25

**CHAPTER 7. PULLING IMAGES FROM A CONTAINER REPOSITORY** ................. 27

7.1. PULLING AN IMAGE .................................................................... 27

7.2. SYNCING IMAGES FROM A CONTAINER REPOSITORY ....................... 27
PREFACE

Use Ansible Builder to create consistent and reproducible automation execution environments for your Red Hat Ansible Automation Platform needs.
MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright’s message.
PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

If you have a suggestion to improve this documentation, or find an error, please contact technical support at https://access.redhat.com to create an issue on the Ansible Automation Platform Jira project using the docs-product component.
CHAPTER 1. INTRODUCTION TO AUTOMATION EXECUTION ENVIRONMENTS

Using Ansible content that depends on non-default dependencies can be complicated because the packages must be installed on each node, interact with other software installed on the host system, and be kept in sync.

Automation execution environments help simplify this process and can easily be created with Ansible Builder.

1.1. ABOUT AUTOMATION EXECUTION ENVIRONMENTS

All automation in Red Hat Ansible Automation Platform runs on container images called automation execution environments. Automation execution environments create a common language for communicating automation dependencies, and offer a standard way to build and distribute the automation environment.

An automation execution environment should contain the following:

- Ansible 2.9 or Ansible Core 2.11-2.15
- Python 3.8-3.11
- Ansible Runner
- Ansible content collections and their dependencies
- System dependencies

1.1.1. Why use automation execution environments?

With automation execution environments, Red Hat Ansible Automation Platform has transitioned to a distributed architecture by separating the control plane from the execution plane. Keeping automation execution independent of the control plane results in faster development cycles and improves scalability, reliability, and portability across environments. Red Hat Ansible Automation Platform also includes access to Ansible content tools, making it easy to build and manage automation execution environments.

In addition to speed, portability, and flexibility, automation execution environments provide the following benefits:

- They ensure that automation runs consistently across multiple platforms and make it possible to incorporate system-level dependencies and collection-based content.
- They give Red Hat Ansible Automation Platform administrators the ability to provide and manage automation environments to meet the needs of different teams.
- They allow automation to be easily scaled and shared between teams by providing a standard way of building and distributing the automation environment.
- They enable automation teams to define, build, and update their automation environments themselves.
- Automation execution environments provide a common language to communicate automation dependencies.
CHAPTER 2. USING ANSIBLE BUILDER

Ansible Builder is a command line tool that automates the process of building automation execution environments by using metadata defined in various Ansible Collections or created by the user.

2.1. WHY USE ANSIBLE BUILDER?

Before Ansible Builder was developed, Red Hat Ansible Automation Platform users could run into dependency issues and errors when creating custom virtual environments or containers that included all of the required dependencies installed.

Now, with Ansible Builder, you can easily create a customizable automation execution environments definition file that specifies the content you want included in your automation execution environments such as Ansible Core, Python, Collections, third-party Python requirements, and system level packages. This allows you to fulfill all of the necessary requirements and dependencies to get jobs running.

2.2. INSTALLING ANSIBLE BUILDER

Prerequisites

- You have installed the Podman container runtime.
- You have valid subscriptions attached on the host. Doing so allows you to access the subscription-only resources needed to install ansible-builder, and ensures that the necessary repository for ansible-builder is automatically enabled. See Attaching your Red Hat Ansible Automation Platform subscription for more information.

Procedure

1. In your terminal, run the following command to activate your Ansible Automation Platform repo:

   ```bash
   # dnf install --enablerepo=ansible-automation-platform-2.4-for-rhel-9-x86_64-rpms ansible-builder
   ```

2.3. BUILDING A DEFINITION FILE

After you install Ansible Builder, you can create a definition file that Ansible Builder uses to create your automation execution environment image. Ansible Builder makes an automation execution environment image by reading and validating your definition file, then creating a Containerfile, and finally passing the Containerfile to Podman, which then packages and creates your automation execution environment image. The definition file that you create must be in yaml format and contain different sections. The default definition filename, if not provided, is execution-environment.yml. For more information on the parts of a definition file, see Breakdown of definition file content.

The following is an example of a version 3 definition file. Each definition file must specify the major version number of the Ansible Builder feature set it uses. If not specified, Ansible Builder defaults to version 1, making most new features and definition keywords unavailable.

```yaml
version: 3
build_arg_defaults: 1
```
ANSIBLE_GALAXY_CLI_COLLECTION_OPTS: `--pre`

dependencies:

  galaxy: requirements.yml

python:
  - six
  - psutil

system: bindep.txt

images:

  base_image:
    name: registry.redhat.io/ansible-automation-platform-24/ee-minimal-rhel9:latest

# Custom package manager path for the RHEL based images

options:

  package_manager_path: /usr/bin/microdnf

additional_build_steps:

  prepend_base:
    - RUN echo This is a prepend base command!

  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/
    # `--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

  prepend_final:
    RUN whoami
    RUN cat /etc/os-release

append_final:

  - RUN echo This is a post-install command!
  - RUN ls -la /etc

1. Lists default values for build arguments.
2. Specifies the location of various requirements files.
3. Specifies the base image to be used. Red Hat support is only provided for the redhat.registry.io base image.
4. Specifies options that can affect builder runtime functionality.
5. Commands for additional custom build steps.

Additional resources:
Additional resources

- For more information about the definition file content, see Breakdown of definition file content.
- To read more about the differences between Ansible Builder versions 2 and 3, see the Ansible 3 Porting Guide.

2.4. BUILDING THE AUTOMATION EXECUTION ENVIRONMENT IMAGE

After you create a definition file, you can proceed to build an automation execution environment image.

Prerequisites

- You have created a definition file.

Procedure

To build an automation execution environment image, run the following from the command line:

```
$ ansible-builder build
```

By default, Ansible Builder looks for a definition file named `execution-environment.yml` but a different file path can be specified as an argument with the `-f` flag:

```
$ ansible-builder build -f definition-file-name.yml
```

where `definition-file-name` specifies the name of your definition file.

2.5. BREAKDOWN OF DEFINITION FILE CONTENT

A definition file is required for building automation execution environments with Ansible Builder, because it specifies the content that is included in the automation execution environment container image.

The following sections breaks down the different parts of a definition file.

2.5.1. Build args and base image

The `build_arg_defaults` section of the definition file is a dictionary whose keys can provide default values for arguments to Ansible Builder. See the following table for a list of values that can be used in `build_arg_defaults`:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSIBLE_GALAXY_CLI_COLLECTION_OPTS</td>
<td>Allows the user to pass arbitrary arguments to the ansible-galaxy CLI during the collection installation phase. For example, the -pre flag to enable the installation of pre-release collections, or -c to disable verification of the server’s SSL certificate.</td>
</tr>
<tr>
<td>ANSIBLE_GALAXY_CLI_ROLE_OPTS</td>
<td>Allows the user to pass any flags, such as –no-deps, to the role installation.</td>
</tr>
</tbody>
</table>
The values given inside `build_arg_defaults` will be hard-coded into the `Containerfile`, so these values will persist if `podman build` is called manually.

**NOTE**

If the same variable is specified in the CLI `--build-arg` flag, the CLI value will take higher precedence.

You can include dependencies that must be installed into the final image in the dependencies section of your definition file.

To avoid issues with your automation execution environment image, make sure that the entries for Galaxy, Python, and system point to a valid requirements file, or are valid content for their respective file types.

### 2.5.1.1. Galaxy

The `galaxy` entry points to a valid requirements file or includes inline content for the `ansible-galaxy collection install -r` command.

The entry `requirements.yml` can be a relative path from the directory of the automation execution environment definition’s folder, or an absolute path.

The content might look like the following:

**Example 2.2. Galaxy entry**

```yaml
collections:
- community.aws
- kubernetes.core
```

### 2.5.1.2. Python

The `python` entry in the definition file points to a valid requirements file or to an inline list of Python requirements in PEP508 format for the `pip install -r` command.

The entry `requirements.txt` is a file that installs extra Python requirements on top of what the Collections already list as their Python dependencies. It may be listed as a relative path from the directory of the automation execution environment definition’s folder, or an absolute path. The contents of a `requirements.txt` file should be formatted like the following example, similar to the standard output from a `pip freeze` command:

**Example 2.3. Python entry**

```bash
boto>=2.49.0
botocore>=1.12.249
pytz
python-dateutil>=2.7.0
awxkit
packaging
requests>=2.4.2
xmltodict
azure-cli-core==2.11.1
```
2.5.1.3. System

The `system` entry in the definition points to a `bindep` requirements file or to an inline list of `bindep` entries, which install system-level dependencies that are outside of what the collections already include as their dependencies. It can be listed as a relative path from the directory of the automation execution environment definition’s folder, or as an absolute path. At a minimum, the the collection(s) must specify necessary requirements for [platform: rpm].

To demonstrate this, the following is an example `bindep.txt` file that adds the `libxml2` and `subversion` packages to a container:

```plaintext
Example 2.4. System entry

libxml2-devel [platform: rpm]
subversion [platform: rpm]
```

Entries from multiple collections are combined into a single file. This is processed by `bindep` and then passed to `dnf`. Only requirements with no profiles or no runtime requirements will be installed to the image.

2.5.2. Images

The `images` section of the definition file identifies the base image. Verification of signed container images is supported with the `podman` container runtime.

See the following table for a list of values that you can use in `images`:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>base_image</code></td>
<td>Specifies the parent image for the automation execution environment which enables a new image to be built that is based on an existing image. This is typically a supported execution environment base image such as <code>ee-minimal</code> or <code>ee-supported</code>, but it can also be an execution environment image that you have created and want to customize further. A <code>name</code> key is required for the container image to use. Specify the <code>signature_original_name</code> key if the image is mirrored within your repository, but is signed with the image’s original signature key. Image names must contain a tag, such as <code>:latest</code>. The default image is <code>registry.redhat.io/ansible-automation-platform-24/ee-minimal-rhel8:latest</code>.</td>
</tr>
</tbody>
</table>

Red Hat Ansible Automation Platform 2.4 Creating and Consuming Execution Environments
NOTE
If the same variable is specified in the CLI `--build-arg` flag, the CLI value will take higher precedence.

2.5.3. Additional build files

You can add any external file to the build context directory by referring or copying them to the `additional_build_steps` section of the definition file. 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 required keys:

**src**

Specifies the source files to copy into the build context directory. This can be an absolute path (for example, `/home/user/.ansible.cfg`), or a path that is relative to the execution environment file. Relative paths can be glob expressions matching one or more files (for example, `files/*.cfg`).

**dest**

Specifies a subdirectory path underneath the `_build` subdirectory of the build context directory that contains the source files (for example, `files/configs`). This can not be an absolute path or contain `..` within the path. Ansible Builder creates this directory for you if it does not already exist.

NOTE

Absolute paths can not include a regular expression. If `src` is a directory, the entire contents of that directory are copied to `dest`.

2.5.4. Additional custom build steps

You can specify custom build commands for any build phase in the `additional_build_steps` section of the definition file. This allows fine-grained control over the build phases.

Use the `prepend_` and `append_` commands to add directives to the `Containerfile` that run either before or after the main build steps are executed. The commands must conform to any rules required for the runtime system.

See the following table for a list of values that can be used in `additional_build_steps`:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prepend_base</td>
<td>Allows you to insert commands before building the base image.</td>
</tr>
<tr>
<td>append_base</td>
<td>Allows you to insert commands after building the base image.</td>
</tr>
<tr>
<td>prepend_galaxy</td>
<td>Allows you to insert before building the galaxy image.</td>
</tr>
<tr>
<td>append_galaxy</td>
<td>Allows you to insert after building the galaxy image.</td>
</tr>
</tbody>
</table>
### Additional Build Steps

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prepend_builder</td>
<td>Allows you to insert commands before building the Python builder image.</td>
</tr>
<tr>
<td>append_builder</td>
<td>Allows you to insert commands after building the Python builder image.</td>
</tr>
<tr>
<td>prepend_final</td>
<td>Allows you to insert before building the final image.</td>
</tr>
<tr>
<td>append_final</td>
<td>Allows you to insert after building the final image.</td>
</tr>
</tbody>
</table>

The syntax for `additional_build_steps` supports both multi-line strings and lists. See the following examples:

**Example 2.5. A multi-line string entry**

```bash
prepend_final: |
    RUN whoami
    RUN cat /etc/os-release
```

**Example 2.6. A list entry**

```bash
append_final:
    - RUN echo This is a post-install command!
    - RUN ls -la /etc
```

#### 2.5.5. Additional resources

- For example definition files for common scenarios, see the Common scenarios section of the Ansible Builder Documentation

#### 2.6. OPTIONAL BUILD COMMAND ARGUMENTS

The `-t` flag will tag your automation execution environment image with a specific name. For example, the following command will build an image named `my_first_ee_image`:

```bash
$ ansible-builder build -t my_first_ee_image
```

**NOTE**

If you do not use `-t` with `build`, an image called `ansible-execution-env` is created and loaded into the local container registry.

If you have multiple definition files, you can specify which one to use by including the `-f` flag:
In this example, Ansible Builder will use the specifications provided in the file named `another-definition-file.yml` instead of the default `execution-environment.yml` to build an automation execution environment image named `another_ee_image`.

For other specifications and flags that you can use with the `build` command, enter `ansible-builder build --help` to see a list of additional options.

### 2.7. CONTAINERFILE

After your definition file is created, Ansible Builder reads and validates it, creates a **Containerfile** and container build context, and optionally passes these to Podman to build your automation execution environment image. The container build occurs in several distinct stages: base, galaxy, builder, and final. The image build steps (along with any corresponding custom prepend_ and append_steps defined in additional_build_steps) are:

1. During the base build stage, the specified base image is (optionally) customized with components required by other build stages, including Python, pip, ansible-core, and ansible-runner. The resulting image is then validated to ensure that the required components are available (as they may have already been present in the base image). Ephemeral copies of the resulting customized base image are used as the base for all other build stages.

2. During the galaxy build stage, collections specified by the definition file are downloaded and stored for later installation during the final build stage. Python and system dependencies declared by the collections, if any, are also collected for later analysis.

3. During the builder build stage, Python dependencies declared by collections are merged with those listed in the definition file. This final set of Python dependencies is downloaded and built as Python wheels and stored for later installation during the final build stage.

4. During the final build stage, the previously-downloaded collections are installed, along with system packages and any previously-built Python packages that were declared as dependencies by the collections or listed in the definition file.

### 2.8. CREATING A CONTAINERFILE WITHOUT BUILDING AN IMAGE

If you are required to use shared container images built in sandboxed environments for security reasons, you can create a shareable **Containerfile**.

```
$ ansible-builder create
```
CHAPTER 3. COMMON AUTOMATION EXECUTION ENVIRONMENT SCENARIOS

Use the following example definition files to address common configuration scenarios.

3.1. UPDATING THE AUTOMATION HUB CA CERTIFICATE

Use this example to customize the default definition file to include a CA certificate to the `additional-build-files` section, move the file to the appropriate directory and, finally, run the command to update the dynamic configuration of CA certificates to allow the system to trust this CA certificate.

**Prerequisites**

- A custom CA certificate, for example `rootCA.crt`.

**NOTE**

Customizing the CA certificate using `prepend_base` means that the resulting CA configuration appears in all other build stages and the final image, because all other build stages inherit from the base image.

```yaml
additional_build_files:
  # copy the CA public key into the build context, we will copy and use it in the base image 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

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

[galaxy]
  server_list = automation_hub
```

3.2. USING AUTOMATION HUB AUTHENTICATION DETAILS WHEN BUILDING AUTOMATION EXECUTION ENVIRONMENTS

Use the following example to customize the default definition file to pass automation hub authentication details into the automation execution environment build without exposing them in the final automation execution environment image.

**Prerequisites**

- You have created an automation hub API token and stored it in a secure location, for example in a file named `token.txt`. 
• Define a build argument that gets populated with the automation hub API token:

```bash
export ANSIBLE_GALAXY_SERVER_AUTOMATION_HUB_TOKEN=$(cat <token.txt>)
```

additional_build_steps:
    prepend_galaxy:
        # 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 host env
        - ARG ANSIBLE_GALAXY_SERVER_AUTOMATION_HUB_TOKEN
        - ENV ANSIBLE_GALAXY_SERVER_LIST=automation_hub
        - ENV ANSIBLE_GALAXY_SERVER_AUTOMATION_HUB_URL=https://console.redhat.com/api/automation-hub/content/<yourhuburl>-synclist/

### 3.3. ADDITIONAL RESOURCES

• For information regarding the different parts of an automation execution environment definition file, see [Breakdown of definition file content](#).

• For additional example definition files for common scenarios, see [Common scenarios section](#) of the [Ansible Builder Documentation](#)
CHAPTER 4. PUBLISHING AN AUTOMATION EXECUTION ENVIRONMENT

4.1. CUSTOMIZING AN EXISTING AUTOMATION EXECUTION ENVIRONMENTS IMAGE

Ansible Controller includes three default execution environments:

- **Ansible 2.9** - Includes Controller modules only
- **Minimal** - Includes the latest Ansible 2.15 release along with Ansible Runner, but does not include collections or other content
- **EE Supported** - Minimal, plus all Red Hat-supported collections and dependencies

While these environments cover many automation use cases, you can add additional items to customize these containers for your specific needs. The following procedure adds the `kubernetes.core` collection to the `ee-minimal` default image:

**Procedure**

1. Log in to `registry.redhat.io` via Podman:

   ```bash
   $ podman login -u="[username]" -p="[token/hash]" registry.redhat.io
   ```

2. Ensure that you can pull the required automation execution environment base image:

   ```bash
   podman pull registry.redhat.io/ansible-automation-platform-24/ee-minimal-rhel8:latest
   ```

3. Configure your Ansible Builder files to specify the required base image and any additional content to add to the new execution environment image.

   a. For example, to add the Kubernetes Core Collection from Galaxy to the image, use the Galaxy entry:

   ```yaml
   collections:
   - kubernetes.core
   ```

   b. For more information about definition files and their content, see the definition file breakdown section.

4. In the execution environment definition file, specify the original `ee-minimal` container’s URL and tag in the `EE_BASE_IMAGE` field. In doing so, your final `execution-environment.yml` file will look like the following:

   **Example 4.1. A customized execution-environment.yml file**

   ```yaml
   version: 3
   images:
     base_image: 'registry.redhat.io/ansible-automation-platform-24/ee-minimal-rhel9:latest'
   dependencies:
   ```
5. Build the new execution environment image by using the following command:

```
$ ansible-builder build -t [username]/new-ee
```

where [username] specifies your username, and new-ee specifies the name of your new container image.

**NOTE**

If you do not use -t with build, an image called ansible-execution-env is created and loaded into the local container registry.

- Use the `podman images` command to confirm that your new container image is in that list:

  **Example 4.2. Output of a podman images command with the image new-ee**

<table>
<thead>
<tr>
<th>REPOSITORY</th>
<th>TAG</th>
<th>IMAGE ID</th>
<th>CREATED</th>
<th>SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>localhost/new-ee</td>
<td>latest</td>
<td>f5509587efbb</td>
<td>3 minutes ago</td>
<td>769 MB</td>
</tr>
</tbody>
</table>

6. Verify that the collection is installed:

```
$ podman run [username]/new-ee ansible-doc -l kubernetes.core
```

7. Tag the image for use in your automation hub:

```
$ podman tag [username]/new-ee [automation-hub-IP-address]/[username]/new-ee
```

8. Log in to your automation hub using Podman:

```
$ podman login -u="[username]" -p="[token/hash]" [automation-hub-IP-address]
```

**NOTE**

You must have admin or appropriate container repository permissions for automation hub to push a container. For more information, see the Manage containers in private automation hub section in Managing content in automation hub.

```
$ podman login -u="[username]" -p="[token/hash]" [automation-hub-IP-address]
```

9. Push your image to the container registry in automation hub:
$ podman push [automation-hub-IP-address]/[username]/new-ee

10. Pull your new image into your automation controller instance:
   a. Go to automation controller.
   b. From the navigation panel, click Administration → Execution Environments.
   c. Click Add.
   d. Enter the appropriate information then click Save to pull in the new image.

   **NOTE**
   If your instance of automation hub is password or token protected, ensure that you have the appropriate container registry credential set up.

4.2. ADDITIONAL RESOURCES (OR NEXT STEPS)

For more details on customizing {ExecEnvNameShort} based on common scenarios, see the following topics in the Ansible Builder Documentation:

- Copying arbitrary files to an execution environment
- Building execution environments with environment variables
- Building execution environments with environment variables and ansible.cfg
CHAPTER 5. POPULATING YOUR PRIVATE AUTOMATION HUB CONTAINER REGISTRY

By default, private automation hub does not include container images. To populate your container registry, you must push a container image to it.

You must follow a specific workflow to populate your private automation hub container registry:

- Pull images from the Red Hat Ecosystem Catalog (registry.redhat.io)
- Tag them
- Push them to your private automation hub container registry

**IMPORTANT**

Image manifests and filesystem blobs were both originally served directly from registry.redhat.io and registry.access.redhat.com. As of 1 May 2023, filesystem blobs are served from quay.io instead. To avoid problems pulling container images, enable outbound connections to the following hostnames:

- cdn.quay.io
- cdn01.quay.io
- cdn02.quay.io
- cdn03.quay.io

Make this change to any firewall configuration that specifically enables outbound connections to registry.redhat.io or registry.access.redhat.com.

Use the hostnames instead of IP addresses when configuring firewall rules.

After making this change you can continue to pull images from registry.redhat.io and registry.access.redhat.com. You do not require a quay.io login, or need to interact with the quay.io registry directly in any way to continue pulling Red Hat container images.

5.1. PULLING IMAGES FOR USE IN AUTOMATION HUB

Before you can push container images to your private automation hub, you must first pull them from an existing registry and tag them for use. The following example details how to pull an image from the Red Hat Ecosystem Catalog (registry.redhat.io).

**Prerequisites**

You have permissions to pull images from registry.redhat.io.

**Procedure**

1. Log in to Podman by using your registry.redhat.io credentials:

   ```bash
   $ podman login registry.redhat.io
   ```
2. Enter your username and password.

3. Pull a container image:

   $ podman pull registry.redhat.io/<container_image_name>:<tag>

**Verification**

To verify that the image you recently pulled is contained in the list, take these steps:

1. List the images in local storage:

   $ podman images

2. Check the image name, and verify that the tag is correct.

**Additional resources**

- See Red Hat Ecosystem Catalog Help for information on registering and getting images.

### 5.2. TAGGING IMAGES FOR USE IN AUTOMATION HUB

After you pull images from a registry, tag them for use in your private automation hub container registry.

**Prerequisites**

- You have pulled a container image from an external registry.
- You have the FQDN or IP address of the automation hub instance.

**Procedure**

- Tag a local image with the automation hub container repository:

  $ podman tag registry.redhat.io/<container_image_name>:<tag> <automation_hub_hostname>/<container_image_name>

**Verification**

1. List the images in local storage:

   $ podman images

2. Verify that the image you recently tagged with your automation hub information is contained in the list.

### 5.3. PUSHING A CONTAINER IMAGE TO PRIVATE AUTOMATION HUB

You can push tagged container images to private automation hub to create new containers and populate the container registry.

**Prerequisites**
You have permissions to create new containers.

You have the FQDN or IP address of the automation hub instance.

**Procedure**

1. Log in to Podman using your automation hub location and credentials:

   ```
   $ podman login -u=<username> -p=<password> <automation_hub_url>
   ```

2. Push your container image to your automation hub container registry:

   ```
   $ podman push <automation_hub_url>/<container_image_name>
   ```

**Troubleshooting**

The *push* operation re-compresses image layers during the upload, which is not guaranteed to be reproducible and is client-implementation dependent. This may lead to image-layer digest changes and a failed push operation, resulting in **Error: Copying this image requires changing layer representation, which is not possible (image is signed or the destination specifies a digest)**.

**Verification**

1. Log in to your automation hub.

2. Navigate to **Container Registry**.

3. Locate the container in the container repository list.
CHAPTER 6. SETTING UP YOUR CONTAINER REPOSITORY

When you set up your container repository, you must add a description, include a README, add groups that can access the repository, and tag images.

6.1. PREREQUISITES TO SETTING UP YOUR CONTAINER REGISTRY

- You are logged in to a private automation hub.
- You have permissions to change the repository.

6.2. ADDING A README TO YOUR CONTAINER REPOSITORY

Add a README to your container repository to provide instructions to your users on how to work with the container. Automation hub container repositories support Markdown for creating a README. By default, the README is empty.

**Prerequisites**

- You have permissions to change containers.

**Procedure**

1. From the navigation panel, select Execution Environments → Execution Environments.
2. Select your container repository.
3. On the Detail tab, click Add.
4. In the Raw Markdown text field, enter your README text in Markdown.
5. Click Save when you are finished.

After you add a README, you can edit it at any time by clicking Edit and repeating steps 4 and 5.

6.3. PROVIDING ACCESS TO YOUR CONTAINER REPOSITORY

Provide access to your container repository for users who need to work with the images. Adding a group allows you to modify the permissions the group can have to the container repository. You can use this option to extend or restrict permissions based on what the group is assigned.

**Prerequisites**

- You have change container namespace permissions.

**Procedure**

1. From the navigation panel, select Execution Environments → Execution Environments.
2. Select your container repository.
3. From the Access tab, click Select a group.
4. Select the group or groups to which you want to grant access and click Next.
5. Select the roles that you want to add to this execution environment and click Next.

6. Click Add.

6.4. TAGGING CONTAINER IMAGES

Tag images to add an additional name to images stored in your automation hub container repository. If no tag is added to an image, automation hub defaults to latest for the name.

Prerequisites
- You have change image tags permissions.

Procedure
1. From the navigation panel, select Execution Environments → Execution Environments.
2. Select your container repository.
3. Click the Images tab.
4. Click the More Actions icon ⋮, and click Manage tags.
5. Add a new tag in the text field and click Add.
6. Optional: Remove current tags by clicking x on any of the tags for that image.
7. Click Save.

Verification
- Click the Activity tab and review the latest changes.

6.5. CREATING A CREDENTIAL IN AUTOMATION CONTROLLER

To pull container images from a password or token-protected registry, you must create a credential in automation controller.

In earlier versions of Ansible Automation Platform, you were required to deploy a registry to store execution environment images. On Ansible Automation Platform 2.0 and later, the system operates as if you already have a container registry up and running. To store execution environment images, add the credentials of only your selected container registries.

Procedure
1. Navigate to automation controller.
2. From the navigation panel, select Resources → Credentials.
3. Click Add to create a new credential.
4. Enter an authorization Name, Description, and Organization.
5. Select the Credential Type.
6. Enter the **Authentication URL**. This is the container registry address.

7. Enter the **Username** and **Password or Token** required to log in to the container registry.

8. Optional: To enable SSL verification, select **Verify SSL**.

9. Click **Save**.
CHAPTER 7. PULLING IMAGES FROM A CONTAINER REPOSITORY

Pull images from the automation hub container registry to make a copy to your local machine. Automation hub provides the podman pull command for each latest image in the container repository. You can copy and paste this command into your terminal, or use podman pull to copy an image based on an image tag.

7.1. PULLING AN IMAGE

You can pull images from the automation hub container registry to make a copy to your local machine.

Prerequisites

- You must have permission to view and pull from a private container repository.

Procedure

1. If are pulling container images from a password or token-protected registry, create a credential in automation controller before pulling the image.
2. From the navigation panel, select Execution Environments → Execution Environments.
3. Select your container repository.
4. In the Pull this image entry, click Copy to clipboard.
5. Paste and run the command in your terminal.

Verification

- Run podman images to view images on your local machine.

7.2. SYNCING IMAGES FROM A CONTAINER REPOSITORY

You can pull images from the automation hub container registry to sync an image to your local machine. To sync an image from a remote container registry, you must first configure a remote registry.

Prerequisites

You must have permission to view and pull from a private container repository.

Procedure

1. From the navigation panel, select Execution Environments → Remote Registries.
2. Add https://registry.redhat.io to the registry.
3. Add any required credentials to authenticate.
Some container registries are aggressive with rate limiting. Set a rate limit under Advanced Options.

4. From the navigation panel, select Execution Environments → Execution Environments.

5. Click Add execution environment in the page header.

6. Select the registry you want to pull from. The Name field displays the name of the image displayed on your local registry.

NOTE

The Upstream name field is the name of the image on the remote server. For example, if the upstream name is set to "alpine" and the Name field is "local/alpine", the alpine image is downloaded from the remote and renamed to "local/alpine".

7. Set a list of tags to include or exclude. Syncing images with a large number of tags is time consuming and uses a lot of disk space.

Additional resources

- See Red Hat Container Registry Authentication for a list of registries.
- See the What is Podman? documentation for options to use when pulling images.
APPENDIX A. AUTOMATION EXECUTION ENVIRONMENTS PRECEDENCE

Project updates will always use the control plane automation execution environments by default, however, jobs will use the first available automation execution environments as follows:

1. The `execution_environment` defined on the template (job template or inventory source) that created the job.

2. The `default_environment` defined on the project that the job uses.

3. The `default_environment` defined on the organization of the job.

4. The `default_environment` defined on the organization of the inventory the job uses.

5. The current `DEFAULT_EXECUTION_ENVIRONMENT` setting (configurable at api/v2/settings/jobs/)

6. Any image from the `GLOBAL_JOB_EXECUTION_ENVIRONMENTS` setting.

7. Any other global execution environment.

**NOTE**

If more than one execution environment fits a criteria (applies for 6 and 7), the most recently created one is used.