Learning Objectives

Learners will be able to…

- **Define package installation**
- **Manage package startup**
- **Modify package configurations**

info

Make Sure You Know

Basics of privilege escalation and sudo/su commands

Privileges

Privilege Escalation

Before we learn to install software, it's important to discuss privileges.

In the Managing Files assignment we defined how to create a file with specific ownership. However, to execute many other tasks, such as creating configuration files in the `/etc` folder, we need `root` user privileges.

Installing software or modifying system configuration files requires privileges escalation. If an ansible process is running from a normal user, the user should have `sudo` privilege escalation rights.

In order to "become" a user with the required privileges for a task or playbook, whether we want to create a file defined for a specific user or execute an action *from* a specific user, the `become` keyword is used.

> info
>
> The `become` keyword leverages various pre-existing privilege escalation tools, such as sudo, su, pfexec, doas, pbrun, dzdo, ksu, runas, machinectl among others.

The Become Directives

The directive `become: true` activates privileges escalation. It can be defined on the level of playbook:

```
- hosts: localhost
  become: true
```

... or task level:

```
tasks:
  - name: Ensure the httpd service is running
    service:
      name: httpd
      state: started
    become: true
```

There are two other **become** directives you should be familiar with:
- `become_user` - set to the user with desired privileges. It defaults to `root` user if none is specified
- This allows you to run Ansible from a novel user and become root before certain operations.
- `become_method` - specifies a method and overrides the `ansible.cfg` default method

```
tasks:
 - name: Ensure the httpd service is running
   service:
     name: httpd
     state: started
   become: true
   become_user: root
```

Install a Package

Package Management

Different operating systems use different package manager systems. Codio is Ubuntu-based and uses its APT to manage software. For RedHat-based distributions `yum` is the primary package manager.

Installing software from a repository can be done using builtin modules for corresponding package systems. Below is a playbook running on an Ubuntu-based OS:

```
- hosts: all
  become: true
  tasks:
    - name: ensure apache2 is at the latest version
        ansible.builtin.apt:
        name: apache2
        state: present
```

- `state` in this example is `present` (default) to install the package. Alternatively, use `latest` to update to latest version if not already installed or `absent` to uninstall the package

Similarly, for RedHat-based systems:

```
- hosts: all
  become: true
  tasks:
    - name: ensure apache2 is at the latest version
        ansible.builtin.yum:
        name: apache2
        state: present
```

Playbook Universality

To make the playbook more universal, `when` can be used to allow it to manage both RedHat (or CentOS) and Ubuntu-based distributions:

```
- hosts: all
  become: true
  tasks:
    - name: ensure Ubuntu apache2 is at the latest version
        ansible.builtin.apt:
        name: apache2
        state: present
      when: ansible_distribution == 'Ubuntu'

    - name: ensure RH apache2 is at the latest version
        ansible.builtin.yum:
        name: apache2
        state: present
      when: ansible_distribution == 'CentOS' or
ansible_distribution == 'RedHat'
```

growthhack

The update_cashe Parameter

`update_cache: yes` is a very useful parameter to perform `apt update` to refresh the package index before installing to avoid ansible failure if an old index is used.

Third-Party Repositories

The APT Repository Module

Not all software is available by default in official repositories, or the version used is outdated; third party repositories are often used to resolve these issues.

The `ansible.builtin.apt_repository` module allows for the management of repositories and signature keys for third-party repositories.

Adding a PPA Repository

Let's say you're using an outdated version of Ubuntu, so you're installing nginx from a third-party repo to get the most up-to-date version. Here's an example of a task that adds a PPA repo:

```
- name: Add nginx stable repository from PPA and install its
signing key on Ubuntu target
  ansible.builtin.apt_repository:
    repo: ppa:nginx/stable
```

Though typically not recommended, repos can also be installed by URL:

```
- name: Add specified repository into sources list using
specified filename
  ansible.builtin.apt_repository:
    repo: deb http://dl.google.com/linux/chrome/deb/ stable main
    state: present
    filename: google-chrome
```

- `repo` defines the APT source string
- `state` adds the repo when `present` (default), or removed when `absent` is used.
- `filename` - Sets the name of the source list file in `sources.list.d`. Extension `.list` is added automatically

## Package Installation From a URL

Software packages can also be installed directly from a URL using `deb` parameter.

```
- name: Install a .deb package from the internet
  ansible.builtin.apt:
    deb: https://example.com/python-ppq_0.1-1_all.deb
```

The `deb` parameter can also be used to install packages from the file system:

```
- name: Install a .deb package from local disk
  ansible.builtin.apt:
    deb: /tmp/python-ppq_0.1-1_all.deb
```

Manage Service State

The Systemd Module

Systemd is an init system that orchestrates Linux services: what needs to be started and when, initializing network configurations, etc.

Most modern Linux distributions use systemd as their service manager. Ansible has matching modules for most common service managers including systemd; the `ansible.builtin.systemd` module is used to restart, reload, stop, and/or update systemd services.

Take a look at the following task that utilizes the systemd module:

```
- name: Make sure a service unit is running
  ansible.builtin.systemd:
    state: started
    name: httpd
```

- `name` is the name of the Systemd unit you want to manage
- `state started` ensures the service is running, if the service is already running Ansible won't do anything
- similarly, `state stopped` will only stop the service if it's not stopped already
- `state restarted` will bounce the unit; it will trigger a restart of services, regardless of current state
- `state reloaded` will also bounce the unit

important

It is recommended that `restarted` and `reloaded` be used in conjunction with handlers to avoid unnecessary restarts.

### The Daemon Reload Parameter

By default, systemd ignores changes to unit (.service) files. If a change has been made to a unit file, such as a change on the command line or changes to environment variables, the parameter `daemon_reload: yes` must be used to reload systemd with the updated file.

Parameter execution order is critical here, so if `daemon_reload` is used with a handler, pay attention to handlers' execution order; reload should come before restart. Otherwise, systemd will pick up the old unit file.

info

For new files...

If a new unit file is created, reload is not needed; the file will be picked up automatically.

To specify the behavior of a service on restart of a unit, use the `enabled` parameter:
- `enabled: true` to enable (start)
- `enabled: false` to disable (stop)

Systemd can start disabled units by means of dependencies or socket activation. To stop the unit from starting it can be masked using `masked: yes`.

Configure Init Services

Managing With Systemd Service Files

While Ansible can manage systemd services, it doesn’t have builtin modules to manage the content of systemd units.

Systemd service files are simple text files, and the `template` or `copy` module can be used to manage them. Although there are no built-in Ansible modules specifically for managing systemd service files, it is possible to manage them by creating a .service file for it.

Below is an example of a unit file that fully controls the behavior of the monitoring system, in this case Grok Exporter. We use the copy module, specify the content, and upload it to `/etc/systemd/system/grok.service`.

```
- name: Copy the Grok Exporter systemd service file
    ansible.builtin.copy:
      content: |-
        [Unit]
        Description=Grok Exporter
        After=network.target

        [Service]
        Type=simple
        User=root
        Group=root
        Nice=-5
        ExecStart=/etc/grok_exporter/grok_exporter -config
/etc/grok_exporter/grok_exporter.conf

        SyslogIdentifier="grok_exporter"
        Restart=always
        StartLimitBurst=1000

        [Install]
        WantedBy=multi-user.target
      dest: /etc/systemd/system/grok.service
      owner: root
      group: root
      mode: 0644
- name: ensure Grok started
  ansible.builtin.systemd:
    name: grok
    enabled: yes
    started: yes
    daemon_reload: yes
```

▼ **The anatomy of the Grok Exporter playbook:**

The first task, `Copy the Grok Exporter systemd service file`, uses the `copy` module contains several parts:

- The [Unit] section includes a description, the defined order, and ensures network connection (internet is required, so this check occurs prior to collecting metrics)
- The [Service] section includes:
  - `Type` can specified for if the service forks, or in this case just to run it and monitor its state
  - the `User` and `Group` from which we are running it
  - `Nice` specifies the priority
  - `ExecStart` includes the command, in this case the Grok Exporter (the binary) that the system should start

    - `Restart:always` means if this binary crashes it will be restarted
    - `StartLimitBurst` specifies a number for which, if exceeded, it will fail
- The [Install] section includes the following parameters:
    - `dest`, the destination, of the package. It's important to note that *all user managed services and system-level packages should be located in `/etc/systemd/system` directory.*
    - `owner` and `group`
    - `mode: 0644` to specify that the owner can read and write, users within the owner's group can read, and all users can read.

The second task in the playbook contains:
- the `name` of the service, in this case `grok`
- `enabled: yes` ensures it starts by default
- `started: yes` indicates we want it started now
- `daemon_reload: yes` means we've created the file, so reload and update

The playbook above exemplifies that we can configure all the services we need by using the simple `copy` (or `template`) of the service file.

Override Default Parameters

Let's imagine we're using Apache2 software and we want to add an additional flag and/or environment variable to the service.

Instead of editing package units in `/usr/lib/systemd/system/`, which might be overridden by a package update, it is recommended to override starting parameters from package default by creating an override file. We create a directory related to the systemd file we want to change, create the override configuration file, and use the `copy` module to override parameters configuration as needed.

We need to ensure the directory for override exists and create the file there:

```
- hosts: localhost
  become: true
  tasks:
    - name: Ensure override httpd exists
        ansible.builtin.file:
          path: /etc/systemd/system/httpd.service.d
          state: directory
          owner: root
          group: root
    - name: Create override for httpd envirnoments
        ansible.builtin.copy:
          content: |-
            Environment=LD_LIBRARY_PATH=/opt/vendor/lib
          dest:
/etc/systemd/system/httpd.service.d/libraries.conf
          owner: root
          group: root
          mode: 0644
        notify:
          - Restart httpd
    - name: ensure httpd started
      ansible.builtin.systemd:
        name: httpd
        enabled: yes
        started: yes
        daemon_reload: yes
  handlers:
    - name: Restart httpd
      ansible.builtin.systemd:
        name: httpd
        restarted: yes
        daemon_reload: yes
```