# Learning Objectives

**Learners will be able to...**

- **Explain control and managed nodes**
- **Define target hosts for Ansible**
- **Construct a simple Ansible playbook**
- **Run an Ansible playbook**

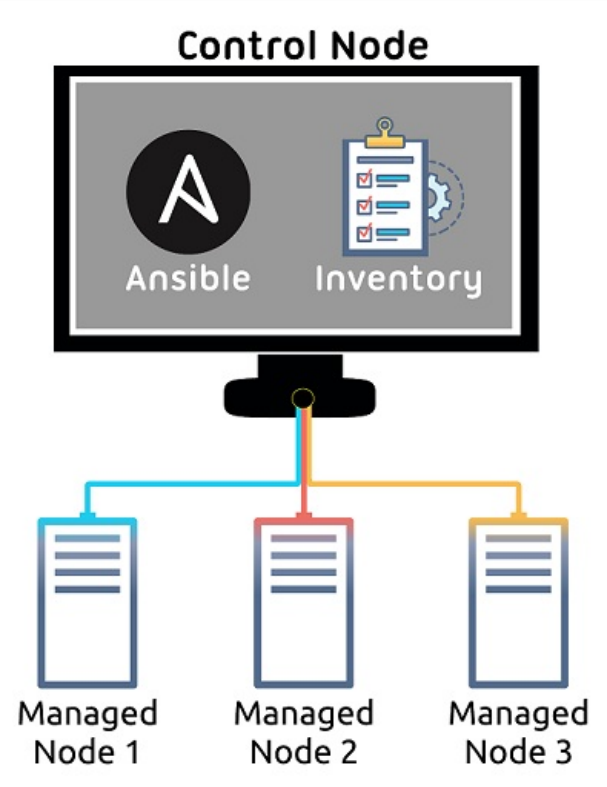
info

## Make Sure You Know

- (some understanding of) Linux/Unix systems and command-line interface
- Basic understanding of system administration concepts
- SSH authentication

# Ansible Architecture

## The Basic Framework




Control node runs Ansible with the inventory file, is connected to all managed nodes

## ### Control Node

The control node is the system (laptop, server, etc.) on which Ansible is installed and runs. You run Ansible commands such as `ansible` or `ansible-inventory` on a control node.

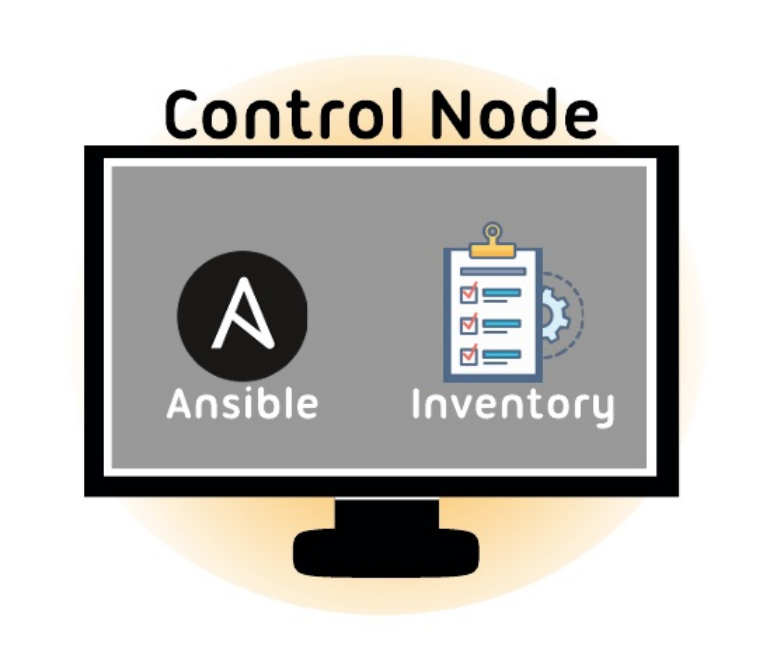
## ### Managed Node (Host)

A managed node refers to a remote system, often referred to as a **host**, that Ansible controls. Ansible's versatility allows it to scale down in order to configure a single managed node, or scale up to configure thousands!

### Inventory

The inventory file, a list of managed nodes that are logically organized, contains all of the needed server information Ansible needs. You create an inventory on the control node to describe host deployments to Ansible.

# Ansible Architecture: Control Node




Control node: runs ansible with the inventory file

## Control Nodes

A control node can be your local computer, CI/CD pipeline, or dedicated control server. Any computer that meets software requirement can be used as control node. There is typically one control node, and sometimes an additional backup control node.

info

### Multiple Control Nodes

It is possible to have multiple control nodes, but be aware: Ansible won't coordinate between them and might run into conflicts when changes are applied from several places at the same time.

In this assignment we are going to use a Codio box as an control node. Ansible is preinstalled in it.

## ### Check Ansible Version

Try running the Ansible command prompt below by copying and pasting it into the terminal:

```
ansible --version
```

Some of the notable information you see displayed here includes:
- The version of Ansible running on the control node
- The versions of Python and Jinja templating engine being used
- The paths for system-level modules and collections (more on this in upcoming lessons)

# Ansible Architecture: Managed Node

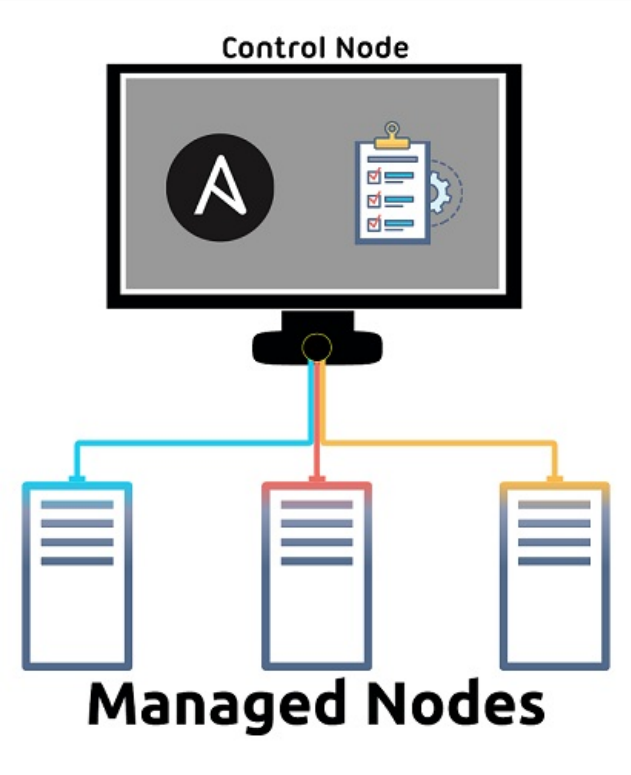



Image of control node and managed nodes; focus on managed nodes

## Managed Nodes

Managed nodes are the target devices we aim to manage. Often referred to as **'hosts'**, managed nodes are often servers but can include network appliances, clients, or any computer.

Ansible requires direct access to any managed nodes from your control node in order to configure them. Since most nodes are behind a NAT firewall, an **SSH** (Secure Shell network protocol) connection must be established. SSH is used to encrypt communications between the control and managed nodes.

SSH also ensure authentication between machines; in general, using Ansible to configure remote hosts *requires* an SSH connection to be established from the control node to manage the other nodes.

important

Managed nodes don’t need to have Ansible installed.

## Combining Managed and Control Nodes

A managed node can be the same as control node; both are running from the same machine so that the machine is provisioning itself. A possible use-case for this would be preparing a golden image for instances on a cloud provider.

Combining a managed and control node together is done if setting up Ansible on the managed node is simpler than on a local machine, due to python version, user access limits, etc.

▼ **More on combining managed and control nodes:**

As an example, let’s say we are building an AMI image on AWS. We will run Ansible on the same machine and it will target `localhost` to configure, meaning the machine will configure itself.

Why would we do this? In a situation where we have Windows users and we want to avoid installing Ansible dependencies, many providers allow Ansible to execute on first launch to provision the server, or run Ansible as an arbitrary script on the first launch.

Additionally, combining managed and control nodes can sometimes provide significant time-savings. Ansible can effectively manage SSH connections, but it can be time consuming due to latency issues establishing and closing connections. Running Ansible from the managed node can bypass the process of opening and closing SSH connections for command executions.

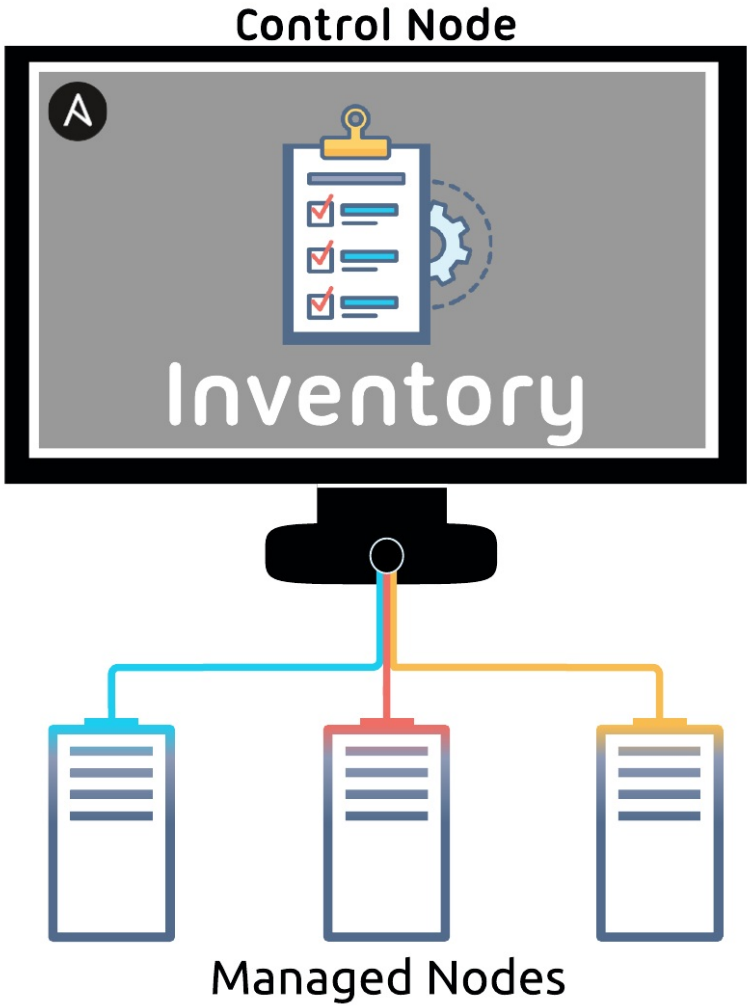
In this case an inventory is not required because you will know what you’re configuring, so you’re simply passing the variables directly.

hackathon

## Real-World Example

Configuring a container instance (managed node) within Codio’s infrastructure from my laptop takes ~40 minutes, but running Ansible locally from the container takes ~7 minutes.

# Ansible Architecture: Inventory




## Inventories

Ansible must be provided with explicit information in order to manage nodes; this information is provided by the inventory file.

The inventory file, also referred to simply as “inventory,” allows you to specify a list of **target hosts** (managed nodes to control). This is the only function the inventory may serve. It can be a static file with node-specific information like an ip address, or it can be automatically created through *service discovery*.

definition

**Service discovery** is the process of automatically detecting devices and services that are available on a network. It is commonly used in microservice architectures and containerization platforms to minimize the configuration effort required by administrators.

▼ **Examples of service discovery:**

An example of service discovery is a containerized application automatically detecting and connecting to a database service without the need for manual configuration.

Another example of service discovery is when a web application needs to connect to a database. With service discovery, the web application can automatically discover the location and connection details of the database, without the need for manual configuration or hardcoding of connection strings. This makes it easier to scale and maintain the application, as changes to the database infrastructure can be automatically reflected in the application's discovery process.

The default location for the inventory file is `/etc/ansible/hosts`, but you can pick a different inventory file with `-i <path>` parameter.

▼ **Is an inventory file required?**

Technically speaking, an inventory file is not required at all! In simple cases, like managing single devices, you can specify inventory hosts in command-line arguments.

However, the power of Ansible lies in its ability to scale up efficiently; using it to manage a single device offers little gain in efficiency over making changes manually.

## Inventory Format

Here's an example of an Ansible inventory file, written in `YAML`:

```
---
all:
  hosts:
    mail.example.com:
  children:
    webservers:
      hosts:
        foo.example.com:
        bar.example.com:
    dbservers:
      hosts:
        one.example.com:
        two.example.com:
        three.example.com:
```

It's important to note that here are two formats for inventory supported by Ansible: `INI` and `YAML`.

Here is the exact same example written in `INI`:

```
mail.example.com

[webservers]
foo.example.com
bar.example.com

[dbservers]
one.example.com
two.example.com
three.example.com
```

▼ **Why INI?**

Ansible is built on `YAML` because of its easy-to-read syntax. So why would we format an inventory in `INI`?

In the majority of cases, using `INI` is not common practice. However, `YAML` and `INI` are very similar in the structures they can define, making `INI` a viable alternative. Using `INI` could be advantageous if you're generating your hosts from a third party source.

growthhack

Here's another situation in which you *might* use `INI`:

Imagine your system is not starting; perhaps nodes are appearing or disappearing due to container service issues. System information may be returned to you in `INI`, and that data can be interpreted by Ansible.

We will be writing our inventory files in YAML for this course as it is the more common practice.

# Inventory: Groups

## Managing Groups

Ansible is often used to manage groups of hosts (managed nodes), which can be defined in a YAML inventory file. By default Ansible creates 2 groups: `all` and `ungrouped`.
- Every host in the inventory is included in the `all` group.
- If a host does not belong to a group other than `all`, it is automatically a member of `ungrouped` as well.
- A host can belong to any number of groups simultaneously, and will always belong to at least two.

In the following example we have defined groups `dbservers` and `webservers`:

```
---
all:
  hosts:
    mail.example.com:
  children:
    webservers:
      hosts:
        foo.example.com:
        bar.example.com:
    dbservers:
      hosts:
        one.example.com:
        two.example.com:
        three.example.com:
```

This example shows the appropriate grouping syntax for an Ansible playbook:
- `children` is used to specify that a host belongs to a particular group: the subgroups `webservers` and `dbservers` are children of `all`.
- *note*: `children` is a reserved word for subgroups
- `hosts` contains the hosts of a particular group. Ex: `foo.example.com` and `bar.example.com` are hosts belonging to the `webservers` subgroup.

## Grouping groups

It is possible to create a child/parent relationship between groups by adding `children:`:

```
---
all:
  children:
    webservers:
      hosts:
        foo.example.com:
        bar.example.com:
    dbservers:
      hosts:
        one.example.com:
        two.example.com:
        three.example.com:
    production:
      children:
        webservers:
```

- Any host that is member of a child group is automatically a member of the parent group (i.e. `webservers` is a member of both `production` *and* `all`).
- Groups can have multiple parents and children, but not circular relationships.
- Hosts can also be in multiple groups, but there will only be one instance of a host at runtime (Ansible merges the data from the multiple groups).

## Ranges of Hosts

We can take advantage of YAML's capabilities to add ranges of hosts. To do so, let's familiarize ourselves with the syntax:
- `[startValue:endValue]` or `[startValue:endValue:increment]`
- including an increment value is optional
- this syntax is valid for numeric and alphabetic ranges
- range values are inclusive

Let's consider the example below:

```
---
all:
  children:
    webservers:
      hosts:
        www[1:7:2].example.com:
    dbservers
      hosts
        www[d:f].example.com
```

- The hosts included in the `webservers` group are `www1.example.com`, `www3.example.com`, `www5.example.com` and `www7.example.com`.
- The hosts included in the `dbservers` group are `wwwd.example.com`, `wwwe.example.com`, and `wwwf.example.com`.

# Inventory: Variables

## Variables

We can also define variables to the inventory by using the keyword `vars`::

```
---
all:
  hosts:
    localhost
  vars:
    foo: bar
```

To help keep our playbooks organized, we can also add an external file of variable definitions using `vars_files`:. For example, let's include the file `sample_vars.yml` from the `vars` folder in the inventory used above:

```
---
all:
  hosts:
    localhost
  vars:
    foo: bar
  vars_files:
    - /vars/sample_vars.yml
```

## Our Inventory File

In the left panel you'll see our inventory file `main.yml`. Let's take a close look at its contents:

- `all`: This is a play definition, which specifies which hosts the playbook should be run on.
- `localhost` is the only member of the `hosts` (the only group belonging to `all`)
- The `vars` section defines the variable used in the playbook:
  - `ansible_connection` specifies the method that Ansible will use to connect to the host. In this case, it is set to local, which means that Ansible will execute commands directly on the localhost
  - `ansible_python_interpreter` specifies the path to the Python interpreter that Ansible will use when executing commands. In this

case, it is set to `{{ansible_playbook_python}}`, which is a variable that is automatically set by Ansible to the path of the Python interpreter that is running the playbook.

# The Playbook

## Anatomy of a Playbook

> definition
>
> ### Play
>
> In Ansible, a **play** is a set of tasks that are executed on a specific set of managed nodes.
>
> An Ansible **playbook** is simply a collection of plays.

Our `hello_world.yml` playbook shown on the left contains only one **play**, which we've named "Ping and print." Right now that play is empty, so let's add **tasks** to it.

> definition
>
> ### Task
>
> In Ansible, a **task** refers to a block of code that performs a single action (copy a file, create a file from a template, start/restart a service, etc.) on managed node(s).
>
> Related tasks are grouped together into a **play** to be executed sequentially on host(s) defined in the inventory.

This play contains two **tasks**:
1. Our first task, "Ping my hosts," checks that our managed nodes can be reached using the `ansible.builtin.ping` **module**.

Here's our first task. Copy and paste it into the playbook below `tasks`:

```
    - name: Ping my hosts
      ansible.builtin.ping:
```

2. Our second task, "Print message," will print "Hello world" during execution using the `ansible.builtin.debug` **module**. Copy and paste

this task below the previous.

```
    - name: Print Message
      ansible.builtin.debug:
        msg: Hellow world
```

definition

## Modules

**Modules** are pre-written scripts that perform specific actions on target hosts. Ansible provides a wide range of built-in modules that cover many common use cases.

▼ **Check your playbook's syntax:**

This is how your playbook should appear for the next exercise (copy and paste this code if needed):

```
- name: Ping and print
  hosts: all
  tasks:
    - name: Ping my hosts
      ansible.builtin.ping:
    - name: Print Message
      ansible.builtin.debug:
        msg: Hellow world
```

## ### Naming Conventions

When naming the play and individual tasks, you should always use descriptive names that make it easy to verify and troubleshoot playbooks. The play used in our example is appropriately named `Ping and print`.

# Run Ansible

## The Basics of Running Ansible

Once we have defined our hosts in the inventory and assemble plays in the playbook, we are ready to run Ansible. To do so we should have an SSH connection configured on the control and managed nodes in order for Ansible to connect to managed nodes.

## Configuring the SSH Connection

We've created our simple inventory list `main.yml`, which contains only a local host node `localhost`.

Run the following command in the terminal to see the list of hosts:

```
ansible all --list-hosts -i inventory/main.yml
```

The following command will check our connection to hosts specified in our `inventory/main.yml` file (run the command and enter 'yes' if the connection has not previously been established):

```
ansible all -m ping -i inventory/main.yml
```

▼ **Host Fingerprint**

When the SSH connection is first established we are notified of the host's fingerprint. In the example here you may have noticed the SHA256 and following fingerprint - this helps establish authenticity with the host.

Every host has a fingerprint. This is a security feature for the authentication process; it validates the host's identity. This prevents 'spoofing', ensuring we do not unknowingly connect to a malicious server.

Now that we have "shaken hands" with `localhost` we can execute our Ansible playbook, `hello_world.yml`.

## Applying a Playbook

Using the `ansible-playbook` command, followed by our inventory and playbook files, we can apply our playbook.

```
ansible-playbook -i inventory/main.yml playbooks/hello_world.yml
```

Let's consider this command's output in the terminal, broken into two sections:
- **Task Status**: We should see the `PLAY [Ping and print]` header followed by three tasks
- **Play Recap**: Where our play results are summarized

## ### Task Status

Under our play "Ping and print" we should see each of our tasks, "Ping my hosts" and "Print message", listed separately; each task has a status of ok which means it ran successfully.

> important
>
> You will see an additional task, listed above the two in our playbook, named "Gathering Facts." This task will be covered extensively on the following page.
>
> For now, just know that the Gathering Facts task is run implicitly by Ansible at the beginning of every play.

## ### Play Recap

The play recap summarizes the results of all tasks in the playbook per host: in our case, for `localhost`.
- `ok=3` indicates that each task ran successfully
- In this example there are three tasks (Gathering Facts, Ping my hosts, and Print message)
- `changed=0` means we did not edit any preexisting files
- `unreachable` informs us if any of our tasks received an error
- `failed` indicates if Ansible is unable to perform the task
- `skipped` would tell us if a task was not executed because it didn't need to be performed
- For example, if a task installs nginx but it's already installed on the host, Ansible will skip the task
- `rescued=0` indicates that no rescue command was performed
- We can add a "rescue" block of code to a task so that if the command returns false, the task will execute the second block of code rather than `fail`

- `ignored` refers to tasks that are told to `ignore_errors`
- Ansible will cease to execute subsequent tasks if a play fails; in this situation,
`ignore_errors = yes` will force Ansible to continue executing tasks

# Ansible Facts

## Facts and Magic Variables

Ansible facts and “magic variables” provide information that can be used in your playbooks and tasks.

Ansible automatically gathers information about hosts specified in the inventory when it runs a playbook. This information, referred to as “facts,” can include operating system and version, the network interfaces and IP addresses, available memory and CPU usage, and much more. This data can be accessed in the variable `ansible_facts`, a dictionary created to store this information.

As stated previously, the Gather Facts task runs implicitly, making facts available to the playbook or task being run, allowing you to reference them in your Ansible code. Based on facts gathered it is possible to tweak the playbooks behavior for different operation systems, releases, versions, etc.

In the tasks shown below, it collects the distribution (code name, description, id). These two tasks are included in the `hostname.yml` playbook.

```
- name: Print My hostname
  ansible.builtin.debug:
    var: ansible_hostname

- name: Print My LSB
  ansible.builtin.debug:
    var: ansible_lsb
```

Run the following command and review its output:

```
ansible-playbook -i inventory/main.yml playbooks/hostname.yml
```

As you can see, the second task returns a dictionary of the Linux Standard Base information.

You can see all available variables in ansible documentation or printing out `ansible_facts`.

Alternatively, magic variables are predefined variables that provide information about the environment, configuration, and execution of a playbook or task. Automatically set by Ansible, they are made available to your playbooks and tasks.

Facts and magic variables are used to make your playbooks and tasks more flexible and adaptable to different environments.