Automation Developer

TEKsystems Global Services

WORKFORCE DEVELOPMENT

Content Usage Parameters

Content refers to material including instructor guides, student guides, lab guides, lab or hands-on activities, computer programs, etc. designed for use in a training program

1

Content is subject to copyright protection

2

Content may only be leveraged by students enrolled in the training program

3

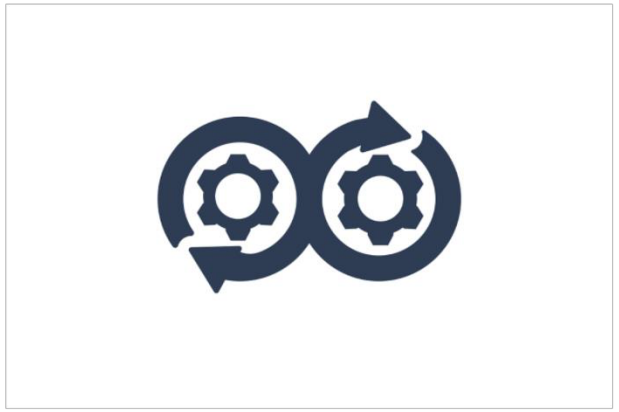
Students agree not to reproduce, make derivative works of, distribute, publicly perform and publicly display in any form or medium outside of the training program

4

Content is intended as reference material only to supplement the instructor-led training

Lab page

https://jruels.github.io/automation-dev/



Defining Variables - Play

You can define different variables for each individual host, or set shared variables for a group of hosts in your inventory. You can also define variables in a play.

```
- hosts: webservers
  vars:
    http_port: 80
```

NOTE: When you define variables in a play, they are only visible to tasks executed in that play.

Defining Variables – External File

You can define variables in reusable variables files and/or in reusable roles. When you define variables in reusable variable files, the sensitive variables are separated from playbooks.
This separation enables storing playbooks in source control and even share them without exposing passwords or other sensitive data.

Defining Variables – External File

This example shows how you can include variables defined in an external file:

```
- hosts: all
  remote_user: root
  vars:
    favcolor: blue
  vars_files:
    - /var/external_vars.yml
```

Defining Variables – External File

The contents of each variables file is a simple YAML dictionary.
For example:

```
---
var1: myfavorite
var2: mysecondfavorite
```

Defining Variables – Command Line

You can define variables when you run your playbook by passing variables at the command line.

--extra-vars (-e)

```
ansible-playbook playbook.yml -extra-vars "red"
```

Defining Variables – Command Line

If you have a lot of special characters, use a JSON or YAML file containing the variable definitions.

```
ansible-playbook release.yml -extra-vars
"@some_var_file.json"
```

Variable Precedence

Ansible does apply variable precedence, and you might have a use for it. Here is the order of precedence from least to greatest (the last listed variables override all other variables):

1. Command line values (for example, -u my_user, these are not variables)
2. role defaults (defined in role/defaults/main.yml)
3. Inventory file or script group vars
4. Inventory group_vars/all
5. Playbook group_vars/all
6. inventory group_vars/*
7. playbook group_vars/*
8. inventory file or script host vars
9. inventory host_vars/*
10. playbook host_vars/*

Variable Precedence

11. host facts / cached set_facts
12. play vars
13. play vars_prompt
14. play vars_files
15. role vars (defined in role/vars/main.yml)
16. block vars (only for tasks in block)
17. task vars (only for the task)
18. include_vars
19. set_facts /registered vars
20. role (and include_role params)
21. include params
22. extra vars (for example –e "user=my_user") – always wins precedence.

Working With Variables - List

A list variable combines a variable name with multiple values. The multiple values can be stored as an itemized list or in square brackets [], separated with commas.

Defining variables as lists:

You can define variables with multiple values using YAML lists. For example:

Region:

- northeast
- southeast
- midwest

Working With Variables - List

Referencing list variables:

When you use variables defined as a list (also called an array), you can use individual, specific fields from that list. The first item in a list is item 0, the second item is item 1.

For example, to select the first element in the array:

```
region: "{{ region[0] }}"
```

POP QUIZ: DISCUSSION

What does this expression return?

```
region: "{{ region[0] }}"
```

POP QUIZ: DISCUSSION

What does this expression return?

```
region: "{{ region[0] }}"
```

- A: northeast
- B: southeast
- C: midwest

POP QUIZ: DISCUSSION

What does this expression return?

```
region: "{{ region[0] }}"
```

- A: northeast
- B: southeast
- C: midwest

Working With Variables - Dictionary

A dictionary stores the data in key-value pairs. Usually, dictionaries are used to store related data, such as the information contained in an ID or a user profile.

You can define more complex variables using YAML dictionaries. A YAML dictionary maps keys to values. For example:

```
region:
  field1: one
  field2: two
```

Working With Variables - Dictionary

When you use variables defined as a key:value dictionary (also called a hash), you can use individual, specific fields from that dictionary using either bracket notation or dot notation:

```
foo['field']
foo.field
```

Dot notation can cause problems because some keys collide with attributes and methods of python dictionaries.

PROTIP: Use bracket notation in most cases.

Registering Variables

You can create variables from the output of an Ansible task with the task keyword `register`. You can use registered variables in any later tasks in your play.

```
- hosts: web_servers
  tasks:
    - name: Register shell command output as variable
      shell: /usr/bin/foo
      register: foo_result
      ignore_errors: true

    - name: Run shell command using output from
previous task
      shell: /usr/bin/bar
      when: foo_result.rc == 5
```

Registering Variables

Registered variables are stored in memory. You cannot cache registered variables for use in future plays. Registered variables are only valid on the host for the rest of the current playbook run.

If a task fails or is skipped, Ansible still registers a variable with a failure or skipped status, unless the task is skipped based on tags.

Querying Nested Data

Many registered variables (and facts) are nested YAML or JSON data structures. You cannot access values from these nested data structures with the simple {{ foo }} syntax. You must use either bracket notation or dot notation. For example, to reference an IP address from your facts using the bracket notation:

```
{{ ansible_facts["eth0"]["ipv4"]["address"] }}
```

Optional Variables

By default, Ansible requires values for all variables in a templated expression. However, you can make specific variables optional.

For example, you might want to use a system default for some items and control the value for others. To make a variable optional, set the default value to the special variable omit:

Optional Variables

In this example, the default mode for the files `/tmp/foo` and `/tmp/bar` is determined by the umask of the system.

Ansible does not send a value for `mode`. Only the third file, `/tmp/baz`, receives the *mode=0444* option.

```
- name: Touch files with optional mode
  ansible.builtin.file:
    dest: "{{ item.path }}"
    state: touch
    mode: "{{ item.mode | default(omit) }}"
  loop:
    - path: /tmp/foo
    - path: /tmp/bar
    - path: /tmp/baz
      mode: "0444"
```

Handlers

Sometimes you want a task to run only when a change is made on a machine. For example, you may want to restart a service if a task updates the configuration of that service, but not if the configuration is unchanged. Ansible uses handlers to address this use case. Handlers are tasks that only run when notified.

POP QUIZ: DISCUSSION

How have you used handlers?

POP QUIZ: DISCUSSION

When should handlers be used?

- When a configuration file is updated, and the service needs to be restarted.
- When a new release is rolled out

Handlers

```
---
- name: Verify Apache installation
  hosts: web
  vars:
    http_port: 80
    max_clients: 200
  tasks::
    - name: Install Apache
    - yum:
    - name: httpd
...
    - name: Apache config
      template:
        src: httpd.j2
        dest: /etc/httpd.conf
      notify:
      - Restart Apache
  handlers:
    - name: Restart Apache
      service:
        name: httpd
        state: restarted
```

Notify Handlers

Tasks can instruct one or more handlers to execute using the `notify` keyword. The `notify` keyword can be applied to a task and accepts a list of handler names that are notified on a task change.

```
tasks:
    name: Template file
    template:
      src: template.j2
      dest: /etc/foo.conf
    notify:
    - Restart apache
    - Restart memcached
```

Notify Handlers

Tasks can instruct one or more handlers to execute using the `notify` keyword. The `notify` keyword can be applied to a task and accepts a list of handler names that are notified on a task change.

```
tasks:
    notify:
    - Restart apache
    - Restart memcached
handlers:
  - name: Restart memcached
    service:
      name: memcached
      state: restarted
  - name: Restart Apache
    service:
      name: apache
      state: restarted
```

Specifying Handlers

Handlers must be named in order for tasks to be able to notify them using the notify keyword.

Alternately, handlers can utilize the listen keyword. Using this handler keyword, handlers can listen on topics that can group multiple handlers as follows:

Specifying Handlers

```
tasks::
    - name: Restart everything
      command: echo "this task will restart the web services"
      notify: "restart web services"

handlers:
  - name: Restart memcached
    service:
      name: memcached
      state: restarted
    listen: "restart web services"

  - name: Restart Apache
    service:
      name: httpd
      state: restarted
    listen: "restart web services"
```

Blocks

Example with Ansible Blocks

```
---
- name: Ansible Blocks
  hosts: server1
  gather_facts: false

  tasks:
    - block:
        - name: List usr directory
          command: "ls -l /usr/"

        - name: List root directory
          command: "ls -l /root"
      become: yes

    - name: List home directory
      command: "ls -l ~/"
```

Recovery

An additional benefit of using ansible blocks is to perform recovery operations.
If any of the tasks within a block fail, the playbook will exit.

With blocks, we can assign a `rescue` block that can contain a bunch of tasks. If any of the tasks within the block fail, the tasks from the recovery block will automatically be executed to perform clean-up activity.

Rescue Variables

Ansible provide a couple of variables for tasks in the `rescue` portion of a block:

- `ansible_failed_task`

  The task that returned 'failed' and triggered the rescue. For example, to get the name use `ansible_failed_task.name`

- `ansible_failed_result`

  The captured return result of the failed task that triggered the rescue. The same as registering the variable.

Rescue Block

Example with Rescue block

```
---
- name: Ansible Blocks
  hosts: server1
  gather_facts: false

  tasks:
    - block:
        - name: List home directory
          command: "ls -l ~/"

        - name: Failing intentionally
          command: "ls -l /tmp/not-home"
      rescue:
       - name: Rescue block (perform recovery)
         debug:
           msg: "Something broke! Cleaning up.."
```

Always Block

An `always` block will be called independent of the task execution status. It can be used to give a summary or perform additional tasks whether the block tasks fail or not.

Always Block

Example with Always block

```
---
- name: Ansible Blocks
  hosts: server1
  gather_facts: false

  tasks:
    - block:
        - name: List home directory
          command: "ls -l ~/"

        - name: Failing intentionally
          command: "ls -l /tmp/not-home"
      always:
       - name: This always executes
         debug:
           msg: "Can't stop me..."
```

Block Practical Example

Now that we've discussed how a block can be used, let's look at practical examples.

- Install, configure, and start a service
- Apply logic to all tasks in the block
- Enable error handling

Block Practical Example

Practical example (Install, configure, and start Apache

```
---
tasks:
  - name: Install, configure, start Apache
    block:
     - name: Install httpd and memcached
       ansible.builtin.yum:
         name:
         - httpd
         - memcached
         state: present
     - name: Apply config template
       ansible.builtin.template:
         src: templates/src.j2
         dest: /etc/template.conf

     - name: Start/enable service
       ansible.builtin.service:
         name: httpd
         state: started
         enabled: true
    when: ansible_facts['distribution'] == 'CentOS'
    become: true
    become_user: root
```

Block Practical Example

The `when` condition evaluated for all tasks in block.

Practical example (Install, configure, and start Apache

```
---
tasks:
  - name: Install, configure, start Apache
    block:
     - name: Install httpd and memcached
       ansible.builtin.yum:
         name:
         - httpd
         - memcached
         state: present
     - name: Apply config template
       ansible.builtin.template:
         src: templates/src.j2
         dest: /etc/template.conf

     - name: Start/enable service
       ansible.builtin.service:
         name: httpd
         state: started
         enabled: true
    when: ansible_facts['distribution'] == 'CentOS'
    become: true
    become_user: root
```

Block Rescue Task Status

If an error occurs in the block and the rescue task succeeds, Ansible reverts the failed status of the original task for the run and continues to run the play as if the original task had succeeded.

The rescued task is considered successful. However, Ansible still reports a failure in the playbook statistics.

Block Handlers

You can use blocks with `flush_handlers` in a rescue task to ensure that all handlers run even if an error occurs:

```
---
tasks:
  - name: Attempt graceful rollback
    block:
     - name: Print a message
       ansible.builtin.debug:
         msg: 'I execute normally'
       changed_when: yes
       notify: run me even after an error

     - name: Force a failure
       ansible.builtin.command: /bin/false
    rescue:
      - name: Make sure all handlers run
        meta: flush_handlers

handlers:
  - name: Run me even after an error
    ansible.builtin.debug:
      msg: 'This handler runs even on error'
```

Lab: Ansible error handling

Jinja2 Templates

Jinja2 templates are simple template files that store variables that can change from time to time. When Playbooks are executed, these variables get replaced by actual values defined in Ansible Playbooks. This way, templating offers an efficient and flexible solution to create or alter configuration file with ease.

Jinja2 Templates

A Jinja2 template file is a text file that contains variables that get evaluated and replaced by actual values upon runtime or code execution. In a Jinja2 template file, you will find the following tags:

`{{ }}` : These double curly braces are the widely used tags in a template file and they are used for embedding variables and ultimately printing their value during code execution.

`{% %}` : These are mostly used for control statements such as loops and if-else statements.

`{# #}` : These denote comments that describe a task.

Jinja2 Templates

In most cases, Jinja2 template files are used for creating files or replacing configuration files on servers.

Apart from that, you can perform conditional statements such as `loops` and `if-else` statements and transform the data using filters and so much more.

Template files have the `.j2` extension, implying that Jinja2 templating is in use.

Jinja2 Templates

A simple Jinja2 template example.

Hey guys! Apache webserver {{ version_number }} is running on {{ server }} Enjoy!

```
The variables are "{{ version_number }}"
and "{{ server }}"
```

Jinja2 Templates

When the playbook is executed, the variables in the template file are replaced with declared vars.

```
---
- hosts:
  vars:
    version_number: "2.3.52"
    server: "Ubuntu"
tasks:
  - name: Jinja 2 template example
    template:
      src: my_template.j2
      dest: /home/ansible/myfile.txt
```

Lab: Ansible templates

Asynchronous Actions & Polling

Ansible runs tasks synchronously, holding the connection to the remote node open until the action is completed. This means within a playbook; each task blocks the next task.

Subsequent tasks will not run until the current task completes.

Challenges:

- Slow
- Long running tasks block all subsequent tasks

Asynchronous Actions & Polling

Playbooks support asynchronous mode and polling, with a simplified syntax.

You can use asynchronous mode in playbooks to avoid connection timeouts or to avoid blocking subsequent tasks. The behavior of asynchronous mode in a playbook depends on the value of poll.

Asynchronous Actions & Polling

For long running tasks, connections to the host can timeout.

If you want to set a longer timeout limit for a certain task in your playbook, use `async` with `poll`.

Ansible will still block the next task in your playbook, waiting until the async task either completes, fails or times out. However, the task will only time out if it exceeds the timeout limit you set with the `async` parameter.

Asynchronous Actions & Polling

To avoid timeouts on a task, specify its maximum runtime and how frequently you would like to poll for status:

```
---
tasks:
    - name: Long running task (15 sec), wait for up to 45 sec, poll every 5 sec
      command: /bin/sleep 15
      async: 45
      poll: 5
```

- `DEFAULT_POLL_INTERVAL`
  The default polling value is 15 seconds.

  There is no default for the async time limit. If you omit the `async` keyword the tasks run synchronously.

  Default async job cache file: `~/.ansible_async`

Asynchronous Actions & Polling

If you want to run multiple tasks in a playbook concurrently, use `async` with `poll` set to 0.

When you set `poll: 0`, Ansible starts the task and immediately moves on to the next task without waiting for a result.

Each async task runs until it either completes, fails or times out (runs longer than its async value). The playbook run ends **without checking back on async tasks**.

Asynchronous Actions & Polling

Playbook with asynchronous task:

```
---
tasks:
    - name: Long running task, allow for 45 sec, fire and forget
      command: /bin/sleep 15
      async: 45
      poll: 0
```

Be careful! Operations that require a lock (yum, apt, etc.) should not be run using `async` if you intend to run other commands later in the playbook on them.

When running with `poll: 0`, Ansible will not automatically cleanup the async job cache file. It will need to be cleaned up manually using the `async_status` module with `mode: cleanup`.

Asynchronous Actions & Polling

Check the status of an `async` task

```
- name: async task
  yum:
    name: docker-io
    state: present
  async: 1000
  poll: 0
  register: yum_sleeper

- name: Check status of async task
  async_status:
    jid: "{{ yum_sleeper.ansible_job_id }}"
  register: job_result
  until: job_result.finished
  retries: 100
  delay: 10
```

Check And Diff Mode

Ansible provides two modes of execution that validate tasks:

- Check mode
  - Ansible runs without making any changes on remote systems.
- Diff mode
  - Ansible provides before-and-after comparisons.

Check And Diff Mode

Check mode is just a simulation. It will not generate output for tasks that use conditionals based on registered variables (results of prior tasks). However, it is great for validating configuration management playbooks that run on one node at a time.

To run an entire playbook in `check` mode:

```
ansible-playbook foo.yml --check
```

Check And Diff Mode

It is also possible to specify that a task always or never runs in check mode regardless of command-line argument.

```
tasks:
  - name: Always change system
    command: /bin/change_stuff --even-in-check-mode
    check_mode: no


- name: Never change system
    lineinfile:
    line: "important config"
    dest: /path/to/config.conf
    state : present
  check_mode: yes
  register: changes_to_important_config
```

Check And Diff Mode

Running single tasks with `check_mode` can be useful for testing Ansible modules, either to test the module itself or to test the conditions under which it would make changes.

Combining `check_mode` and `register` provides even more detail on potential changes.

Check And Diff Mode

It is possible to skip a task or ignore errors when using `check_mode` by specifying `ansible_check_mode` boolean

```
tasks:
  - name: Skip in check mode
    git:
      repo: ssh://git@github.com/me/hello-world.git
      dest: /home/me/hello-world
    when: not ansible_check_mode

  - name: Ignore errors in check mode
    git:
      repo: ssh://git@github.com/me/hello-world.git
      dest: /home/me/hello-world
    ignore_errors: "{{ ansible_check_mode }}"
```

Check And Diff Mode

The `--diff` option for `ansible-playbook` can be used with `--check` or alone.

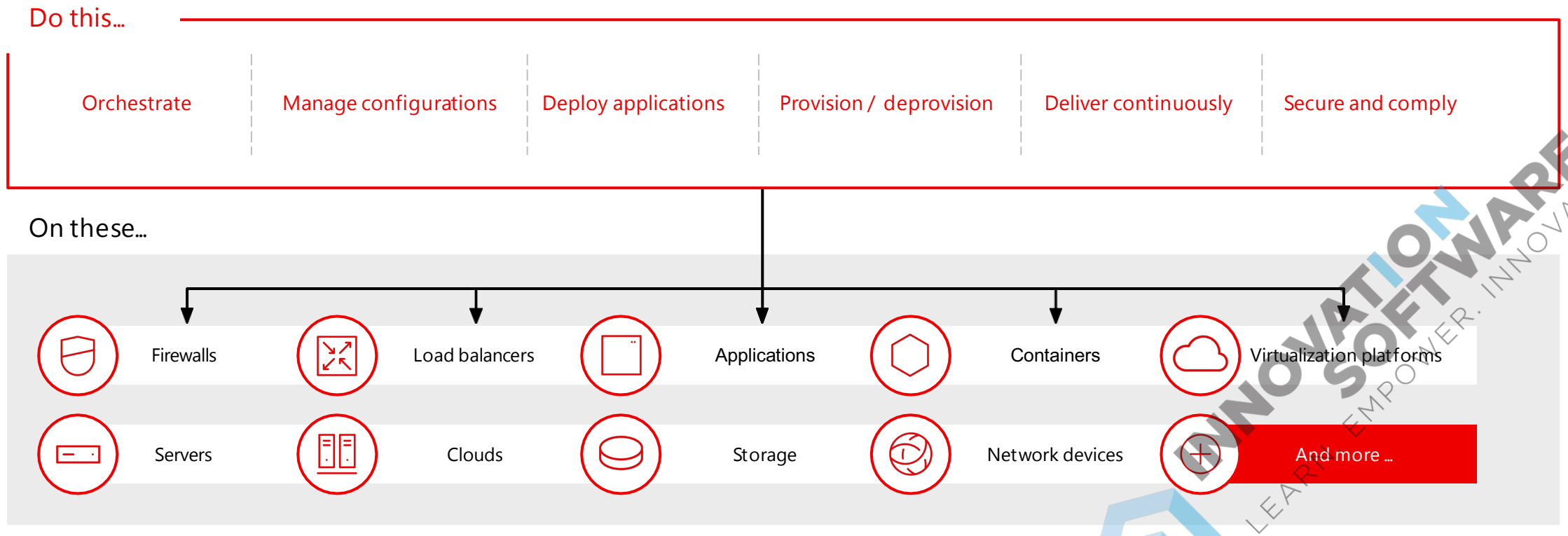
When you run in diff mode, any module that supports diff mode reports the changes made or, if used with `--check`, the changes that would have been made.

Diff mode is most common in modules that manipulate files (for example, the `template` module) but other modules might also show 'before and after' information (for example, the `user` module).

Why The Ansible Automation Platform?

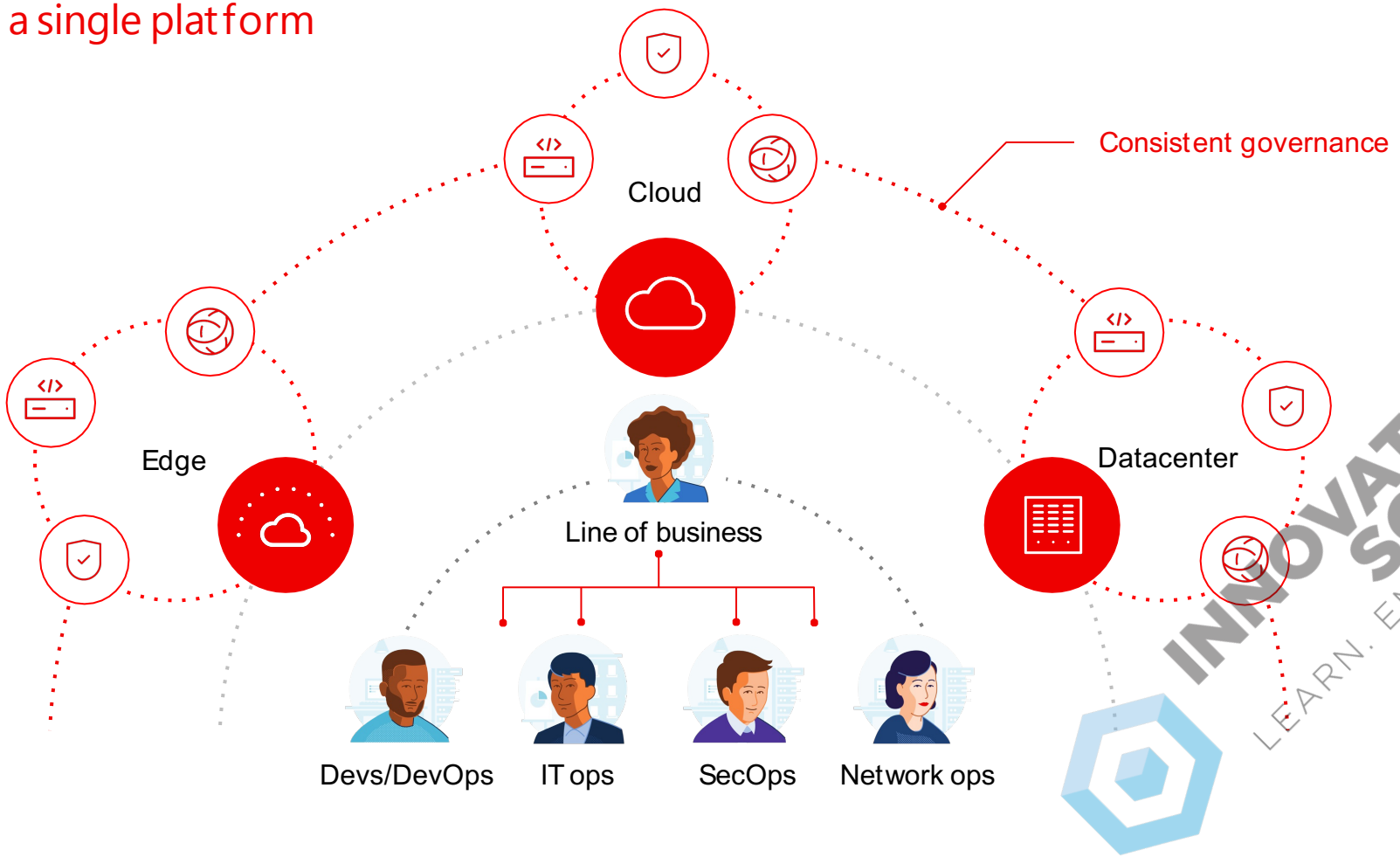
Automate the deployment and management of automation

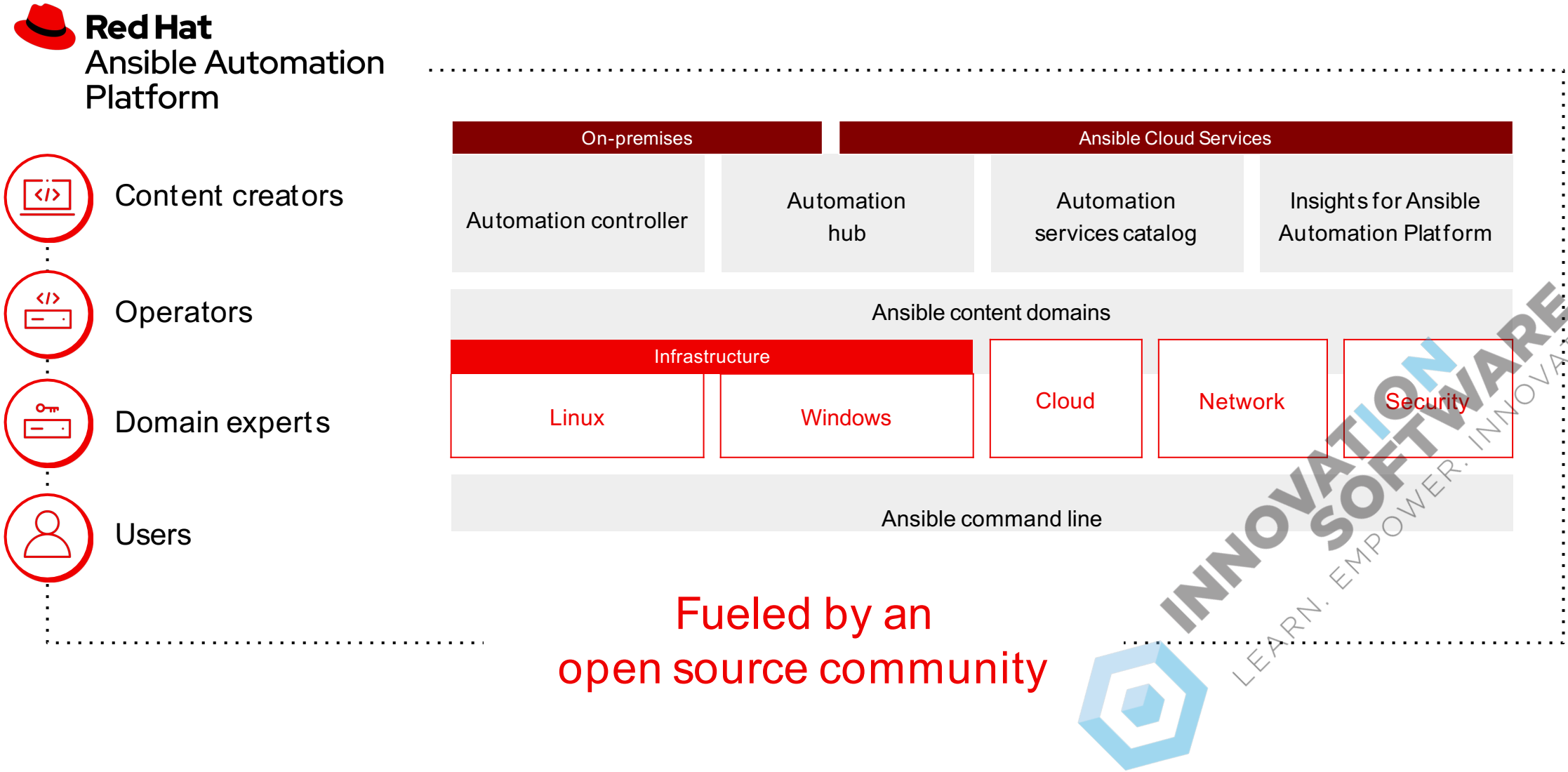
Your entire IT footprint



Break Down Silos

Different teams a single platform





Automation Platform Concepts

A control node is installed with Ansible and is used to run playbooks.

- Contains the Ansible Engine software, the playbook, and it's supporting files.
- Red Hat Automation Platform is a control node that also provides a central web interface, authentication, and API for Ansible.

A managed host is a machine that is managed by Ansible automation.

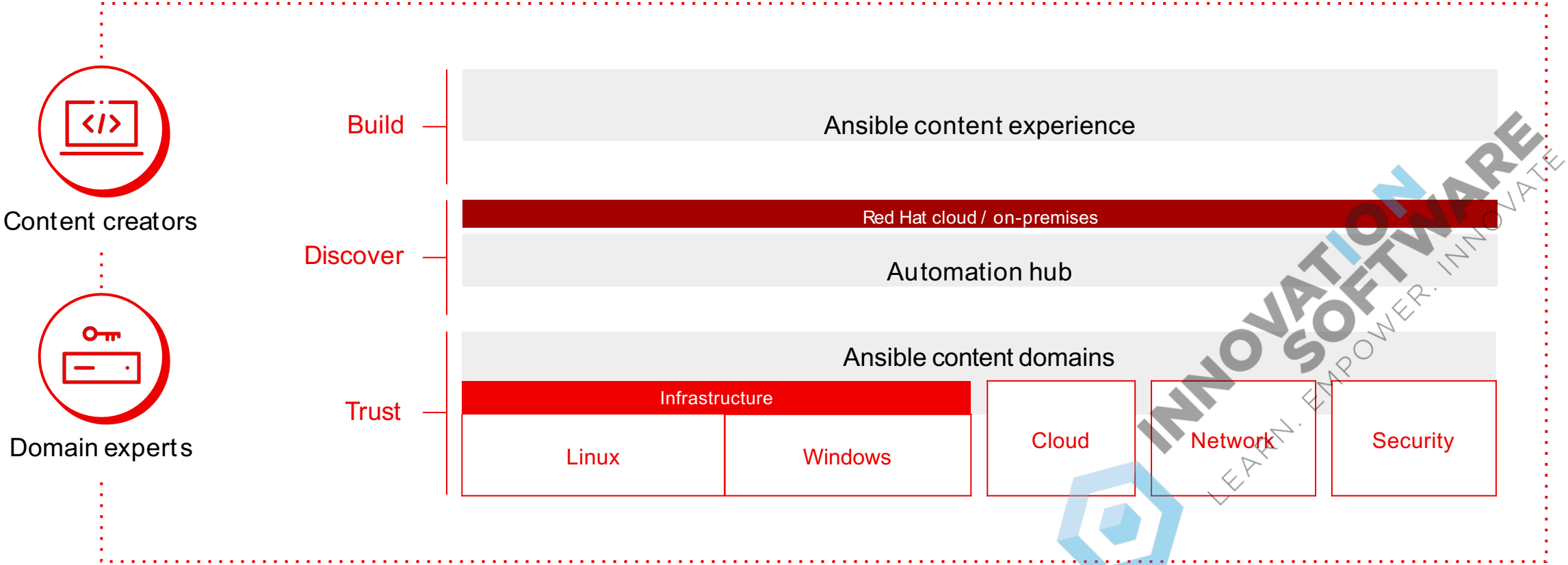
- Does not have Ansible installed
- Does need to be configured to allow Ansible to connect to the host
- Must be listed in the inventory (or generated by a dynamic inventory script or plugin)

Ansible Automation Platform Infrastructure

Create Code

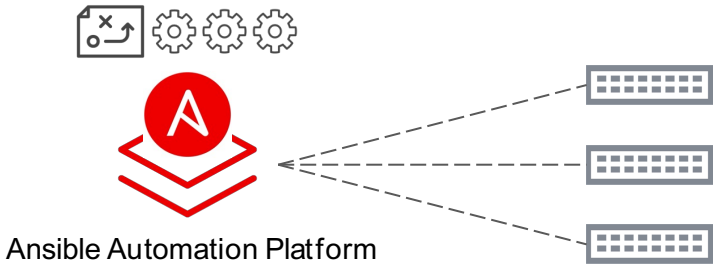
Create

The automation lifecycle



How Ansible Automation Works

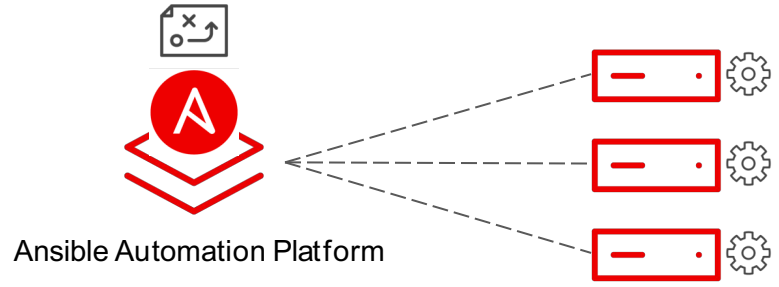
Module code is executed locally on the control node



Local Execution

Network Devices / API Endpoints

Module code is copied to the managed node, executed, then removed



Remote Execution

Linux / Windows Hosts

Installation Architecture Options

Red Hat Ansible Automation Platform can be implemented using one of the following architectures:

- Single Machine with Integrated Database
- Single Machine with Remote Database
- Multi Machine Cluster with Remote Database
- OpenShift Pod with Remote Database

Lab Environment

This class uses a single-node installation with an integrated database:

- Need a system installed with Red Hat Enterprise Linux (bare metal, virtual machine, or cloud instance)
  - Cloud VM
  - 2 vCPU / 4 GB RAM / at least 40GB of storage

Sign up for an evaluation of Automation Platform from Red Hat:

- https://www.redhat.com/en/technologies/management/ansible/try-it

Installer

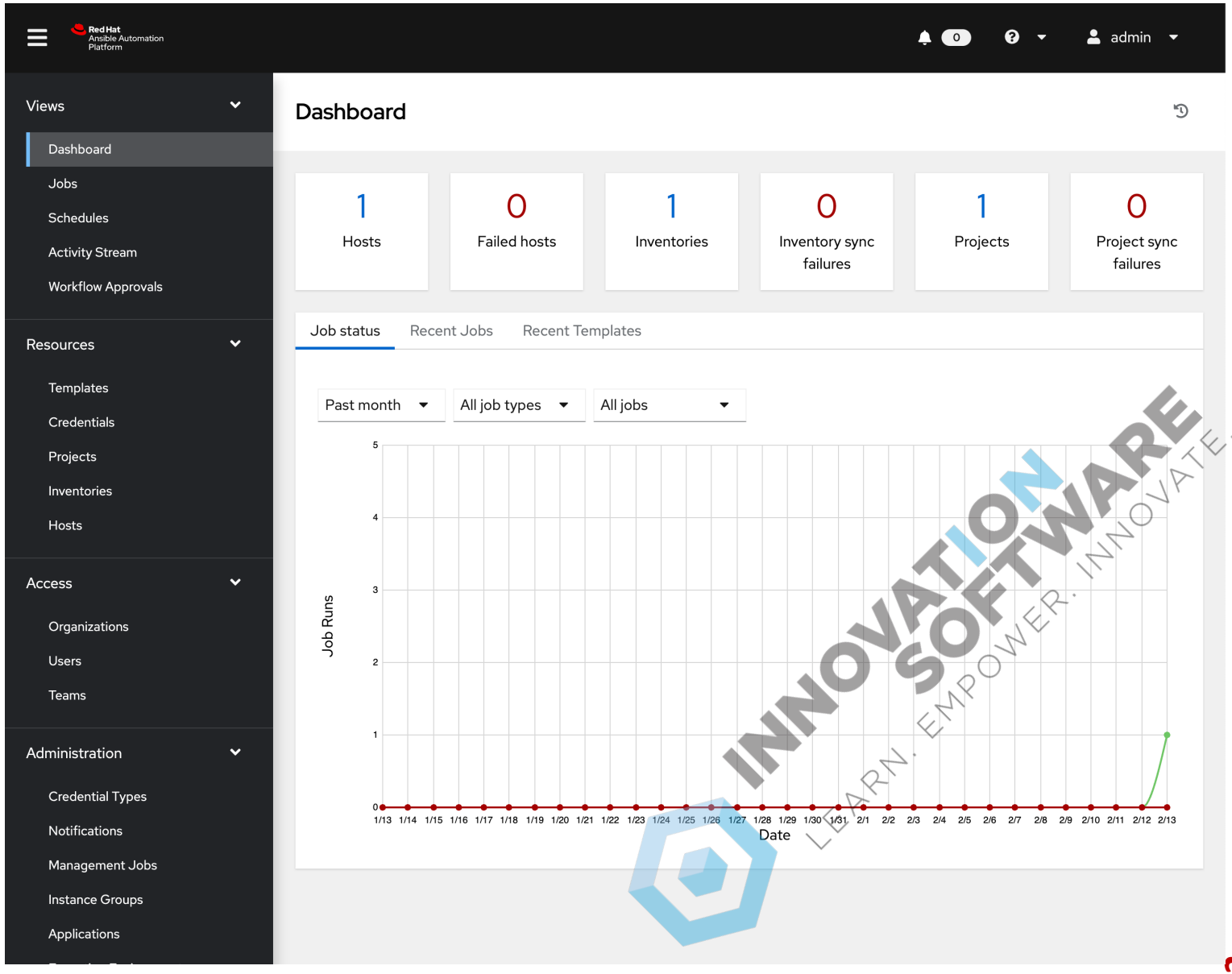
Two different installation packages are available for Automation Platform:

- Standard setup
  - https://access.redhat.com/downloads/content/480
  - Requires internet connectivity to download Automation Platform packages from repositories.

- Bundled installer
  - Download from same link as “Standard setup”, but choose the packages with “Bundle” in the name.
  - Includes initial RPM packages for Automation Platform
  - May be installed on systems without internet access.

Automation Platform Dashboard

- The main control center for Red Hat Ansible Tower.
- Displayed when you log in.
- Composed of four reporting sections:
  - Summary
  - Job Status
  - Recently Used
  - TemplatesRecent Job Runs



Dashboard Navigation

The dashboard contains links to common resources.

Organizations	Control what resources are visible to which users.
Teams	Groups of users that need to access the same resources
Users	User accounts
Jobs	A history of previous Ansible runs.
Templates	Prepared playbooks settings that can be “launched” to run a job.
Credentials	Authentication secrets for managed hosts and Git repositories
Projects	Sources of Ansible Playbooks (usually Git repo)
Inventories	Inventories of managed hosts
Inventory Scripts	Dynamic inventory
Notifications	Configurable for job completion or failure
Management Jobs	Special jobs used to maintain Ansible Platform.

Lab: Install Ansible Automation Platform