Ansible Automation

TEKsystems Global Services

WORKFORCE DEVELOPMENT

ACCESSING ANSIBLE DOCS

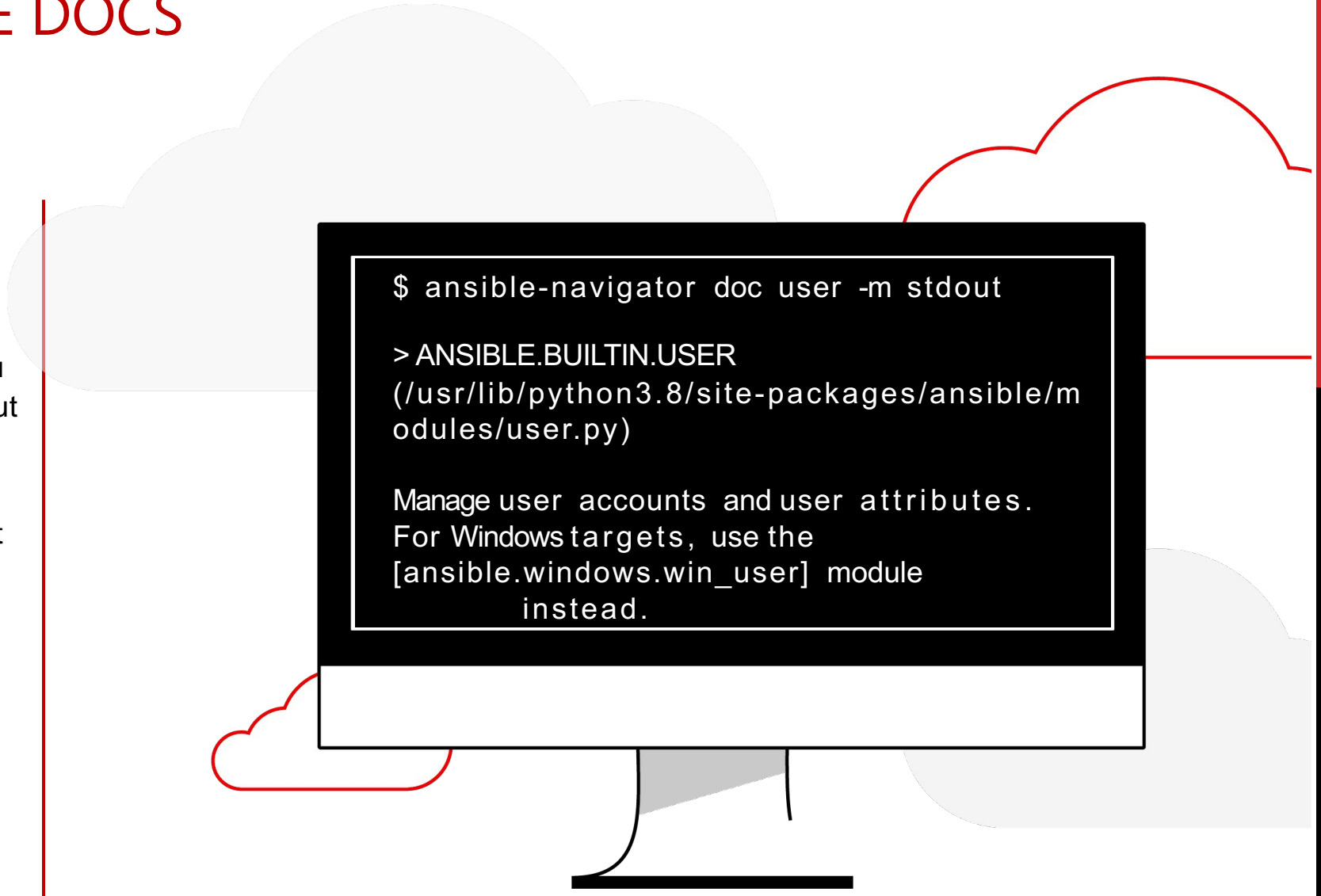
With the use of the latest command utility ansible-navigator, one can trigger access to all the modules available to them as well as details on specific modules.

A formal introduction to ansible-navigator and how it can be used to run playbooks in the following exercise.

ACCESSING ANSIBLE DOCS

Aside from listing a full list of all the modules, you can use ansible-navigator to provide details about a specific module.

In this example, we are getting information about the user module.



ANSIBLE PLAYBOOK VARIABLES

```yaml
---
- name: variable playbook test
  hosts: localhost

  vars:
    var_one: awesome
    var_two: ansible is
    var_three: "{{ var_two }} {{ var_one }}"

  tasks:
    - name: print out var_three
      debug:
        msg: "{{ var_three }}"
```

ANSIBLE PLAYBOOK VARIABLES


```
---
- name: variable playbook test
  hosts: localhost

  vars:
    var_one: awesome
    var_two: ansible is
    var_three: "{{ var_two }} {{ var_one }}"

  tasks:
    - name: print out var_three
      debug:
        msg: "{{ var_three }}"
```

ansible is awesome

ANSIBLE FACTS

- Just like variables, really...
- ..but: coming from the host itself!
- Check them out with the setup module

ANSIBLE FACTS

Ansible playbooks

```
---
- name: facts playbook
  hosts: localhost

  tasks:
    - name: Collect all facts of host
      setup:
        gather_subset:
          - 'all'
```

```
$ ansible-navigator run playbook.yml
```

ANSIBLE NAVIGATOR TUI

Ansible Navigator TUI

```
PLAY NAME              OK          UNREACHABLE    FAILED    SKIPPED    IGNORED    IN PROGRESS    TASK COUNT    PROGRESS
0 | facts playbook        CHANGE          0          0          0          0              0             2    COMPLETE
                       D2        0

RESULT      HOST                NUMBER    CHANGED    TASK                          TASK ACTION          DURATION
0 | OK      localhost                0      False    Gathering Facts               gather_facts               1s
1 | OK      localhost                1      False    Collect all facts of host     setup                      1s

PLAY [facts playbook:1]
*********************************************************************************************************************

TASK [Collect all facts of host]
*********************************************************************************************************************
OK: [localhost]
.
.
12 |   ansible_facts:
 13|     ansible_all_ipv4_addresses:
 14|     - 10.0.2.100
 15|     ansible_all_ipv6_addresses:
 16|     - fe80::1caa:f0ff:fe15:23c4
```

CONDITIONALS VIA VARS

Example of using a variable labeled *my_mood* and using it as a conditional on a particular task.

CONDITIONALS VIA VARS


```
---
- name: variable playbook test
  hosts: localhost

  vars:
    my_mood: happy

  tasks:
    - name: task, based on my_mood var
      debug:
        msg: "Yay! I am {{ my_mood }}!"
      when: my_mood == "happy"
```

Alternatively

```
    - name: task, based on my_mood var
      debug:
        msg: "Ask at your own risk. I'm {{ my_mood }}!"
      when: my_mood == "grumpy"
```

CONDITIONALS WITH FACTS

Ansible Conditionals w/ Facts

```
---
- name: variable playbook test
  hosts: localhost

  tasks:
  - name: Install apache
    apt:
      name: apache2
      state: latest
    when: ansible_distribution == 'Debian' or
          ansible_distribution == 'Ubuntu'

  - name: Install httpd
    yum:
      name: httpd
      state: latest
    when: ansible_distribution == 'RedHat'
```

TASK STATE

Using Previous Task State

```
---
- name: variable playbook test
  hosts: localhost

  tasks:
  - name: Ensure httpd package is present
    yum:
      name: httpd
      state: latest
    register: http_results

  - name: Restart httpd
    service:
      name: httpd
      state: restart
    when: httpd_results.changed
```

VARIABLES AND LOOPS


```yaml
---
- name: Ensure users
  hosts: node1
  become: yes

  tasks:
    - name: Ensure user is present
      user:
        name: dev_user
        state: present

    - name: Ensure user is present
      user:
        name: qa_user
        state: present

    - name: Ensure user is present
      user:
        name: prod_user
        state: present
```

VARIABLES AND LOOPS

Ansible Variables & Loops

```
---
- name: Ensure users
  hosts: node1
  become: yes

  tasks:
    - name: Ensure user is present
      user:
        name: "{{item}}"
        state: present
      loop:
        - dev_user
        - qa_user
        - prod_user
```

RUNNING A PLAYBOOK

Using the latest ansible-navigator command

What is ansible-navigator?

ansible-navigator command line utility and text-based user interface (TUI) for running and developing Ansible automation content.

It replaces the previous command used to run playbooks “ansible-playbook”.

ANSIBLE NAVIGATOR

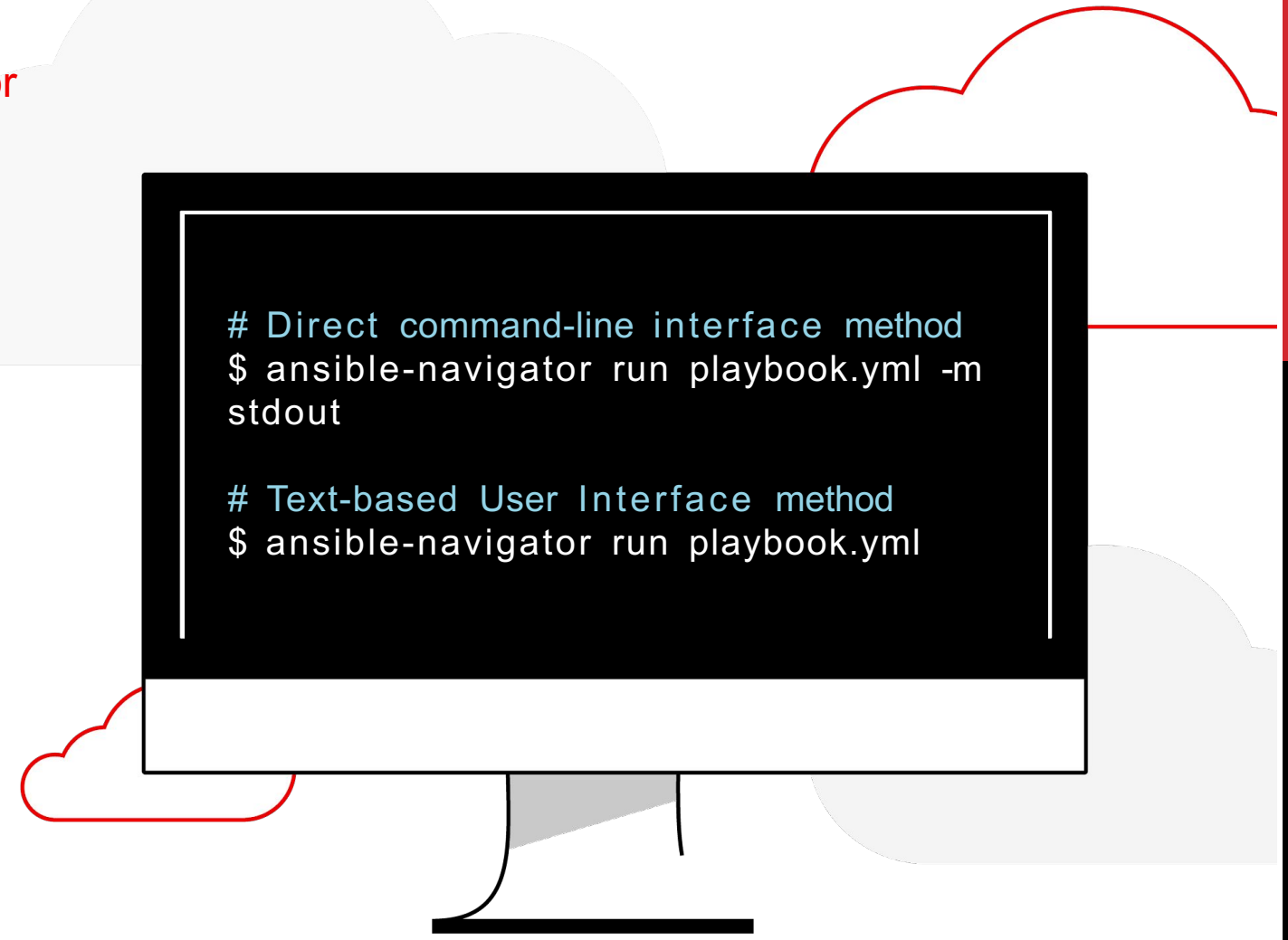
Bye ansible-playbook, Hello ansible-navigator

How do I use ansible-navigator?

As previously mentioned, it replaces the ansible-playbook command.

As such it brings two methods of running playbooks:

- Direct command-line interface
- Text-based User Interface (TUI)



ANSIBLE NAVIGATOR

Mapping to previous Ansible commands

ansible command	ansible-navigator command
ansible-config	ansible-navigator config
ansible-doc	ansible-navigator doc
ansible-inventory	ansible-navigator inventory
ansible-playbook	ansible-navigator run

ANSIBLE NAVIGATOR

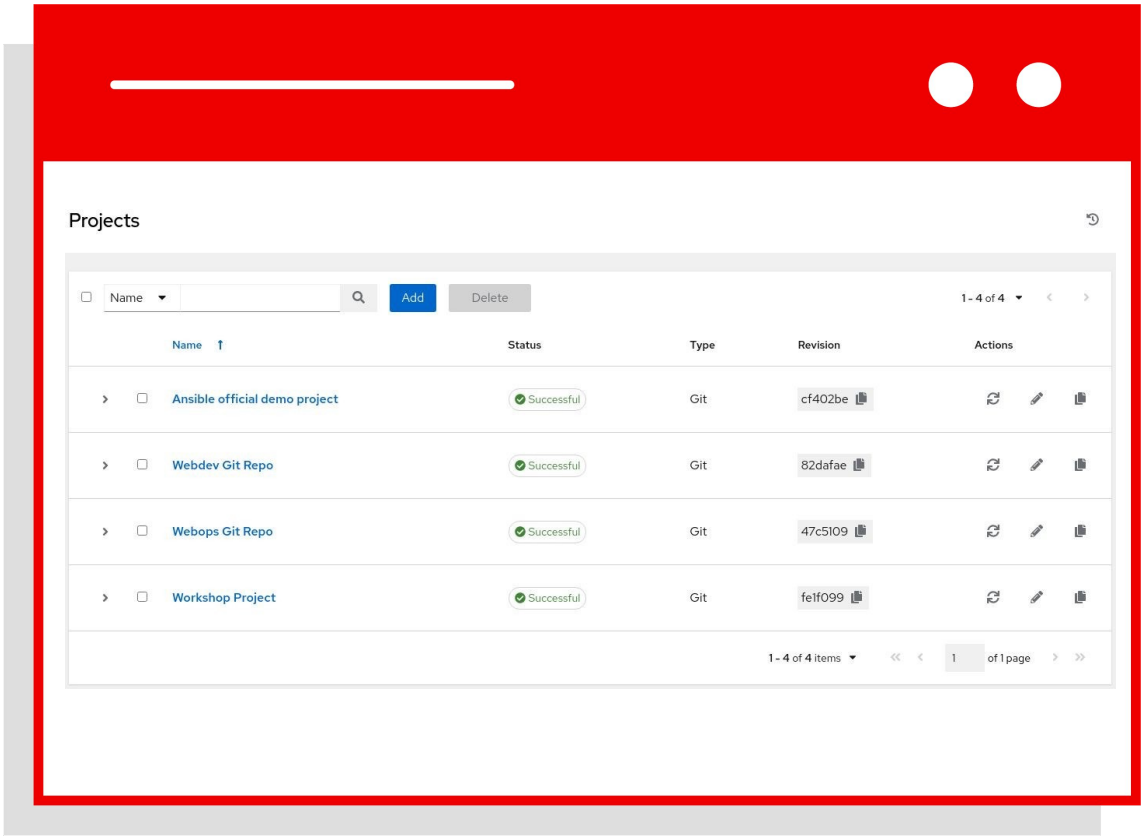
Common subcommands

Name	Description	CLI Example	Colon command within TUI
collections	Explore available collections	ansible-navigator collections --help	:collections
config	Explore the current ansible configuration	ansible-navigator config --help	:config
doc	Review documentation for a module or plugin	ansible-navigator doc --help	:doc
images	Explore execution environment images	ansible-navigator images --help	:images
inventory	Explore and inventory	ansible-navigator inventory --help	:inventory
replay	Explore a previous run using a playbook artifact	ansible-navigator replay --help	:replay
run	Run a playbook	ansible-navigator run --help	:run
welcome	Start at the welcome page	ansible-navigator welcome --help	:welcome

PROJECT

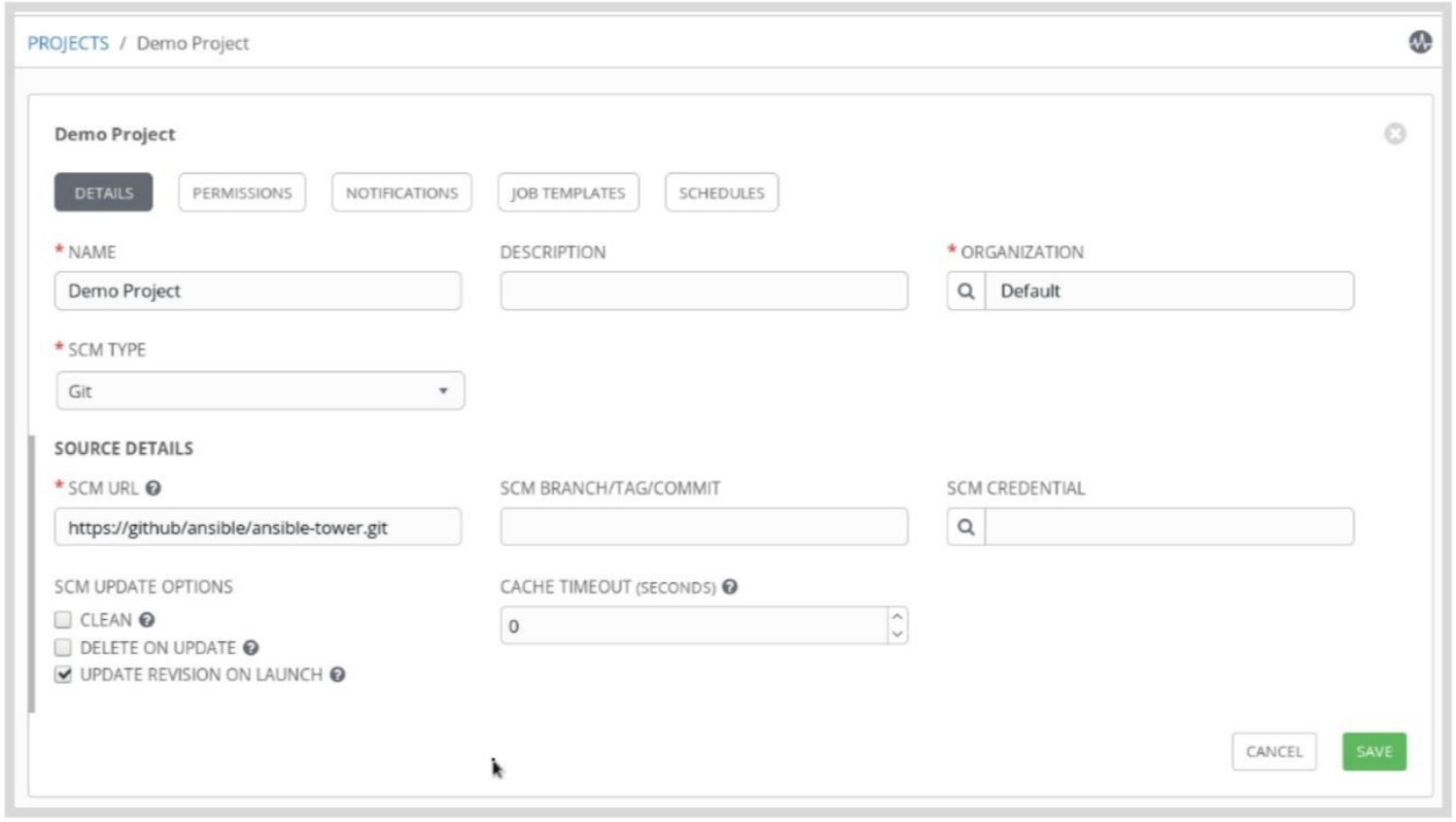
A project is a logical collection of Ansible Playbooks, represented in Ansible Automation Controller.

You can manage Ansible Playbooks and playbook directories by placing them in a source code management system supported by Ansible Tower, including Git, Subversion, and Mercurial.



PROJECT

- Under **Projects** in the left navigation bar, an example project named **Demo Project** is displayed.
- This project is configured to get Ansible project materials, including a playbook, from a public Git repository.
- You can also prepare a credential so you can access a private Git repository that needs authentication.



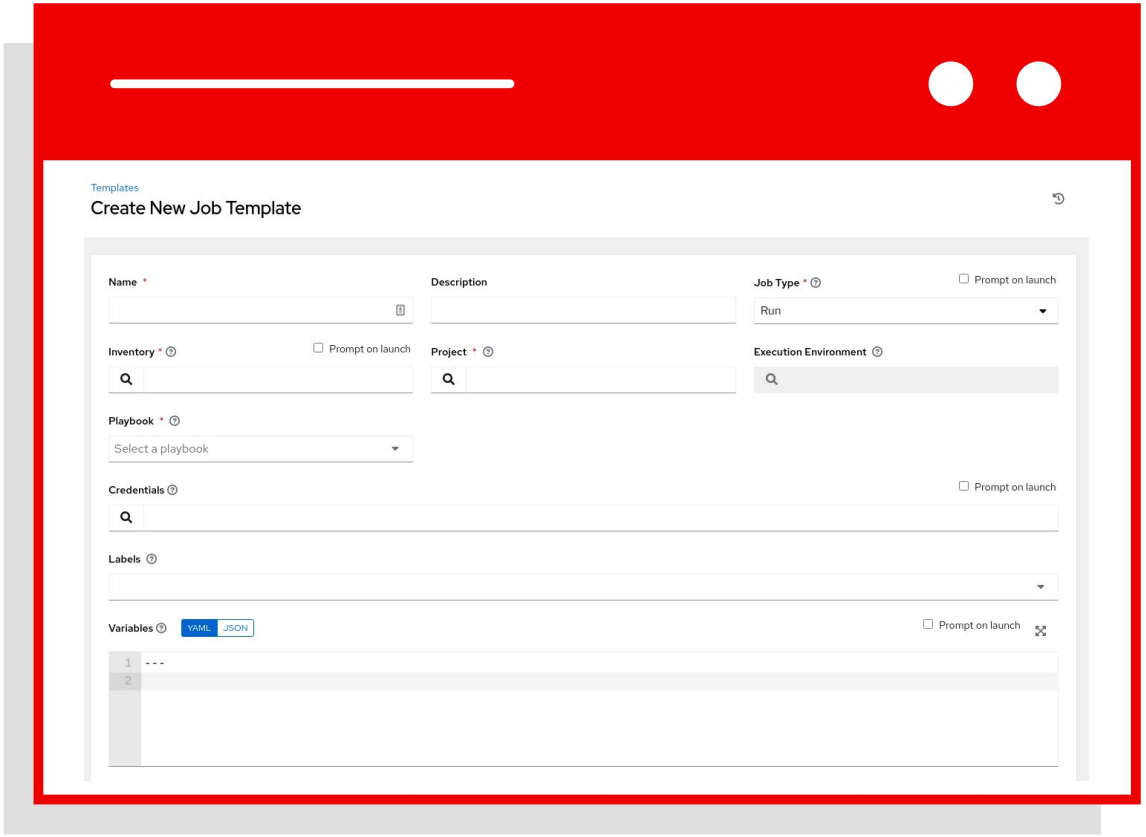
JOB TEMPLATES

Everything in Ansible Tower revolves around the concept of a **Job Template**. Job Templates allow Ansible Playbooks to be controlled, delegated and scaled for an organization.

Job templates also encourage the reuse of Ansible Playbook content and collaboration between teams.

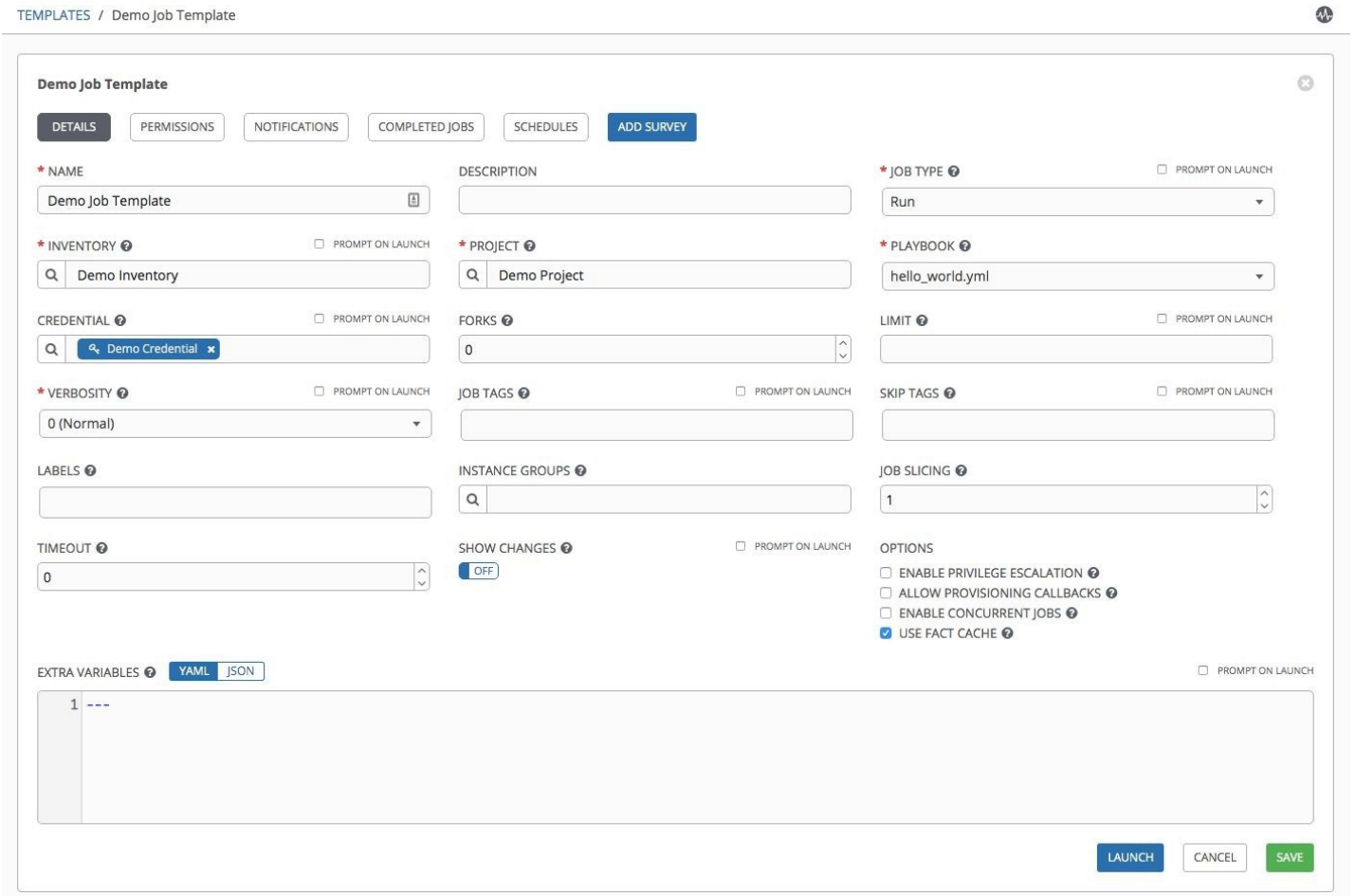
A **Job Template** requires:

- An **Inventory** to run the job against
- A **Credential** to login to devices.
- A **Project** which contains Ansible Playbooks



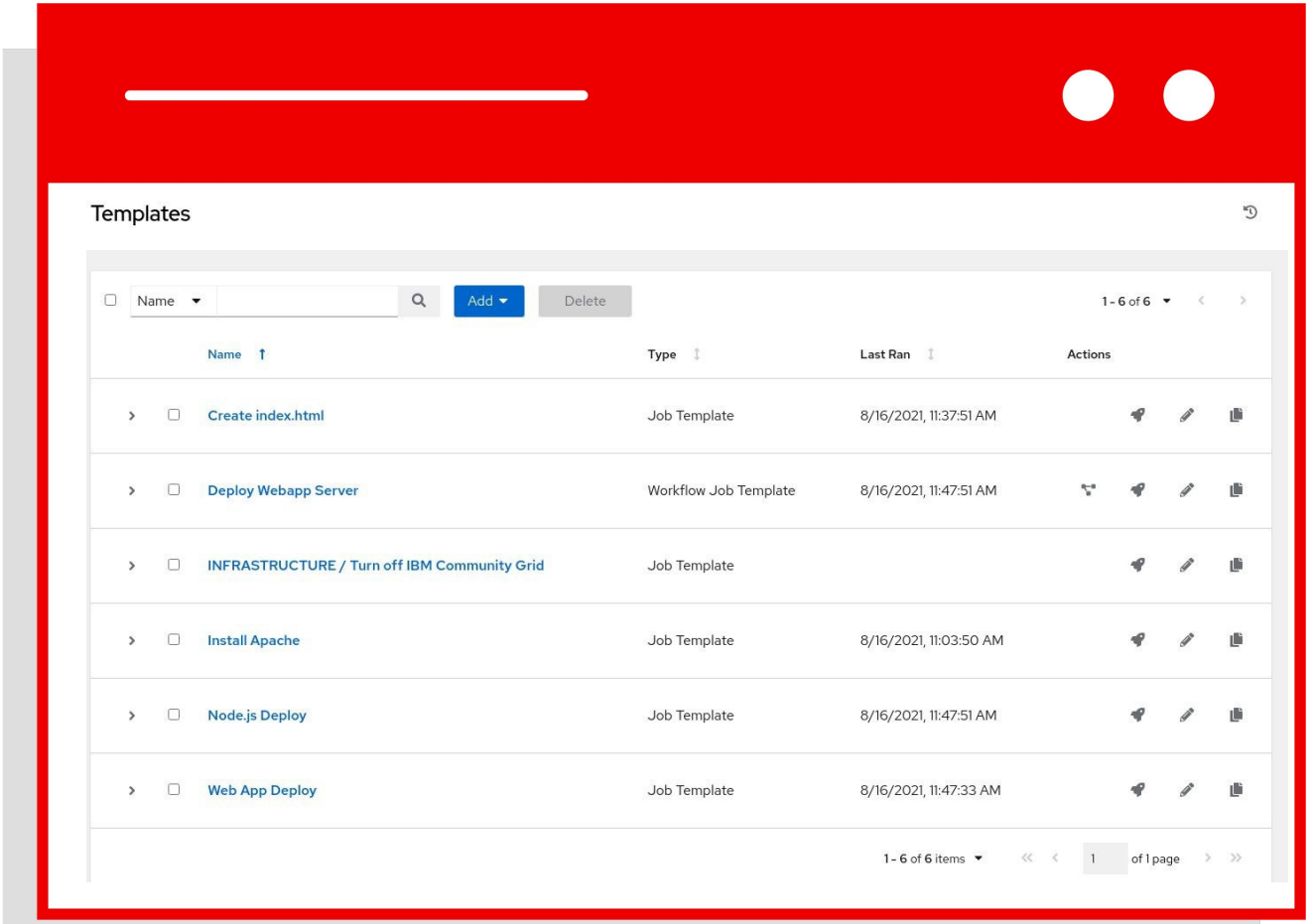
JOB TEMPLATES

- Under **Templates** in the left navigation bar, an example template called **Demo Job Template** is displayed.
- This job template runs the hello_world.yml playbook from **Demo Project** on the hosts in **Demo Inventory**, using **Demo Credential** to authenticate access.
- This initial job template can be used to test Ansible Tower.



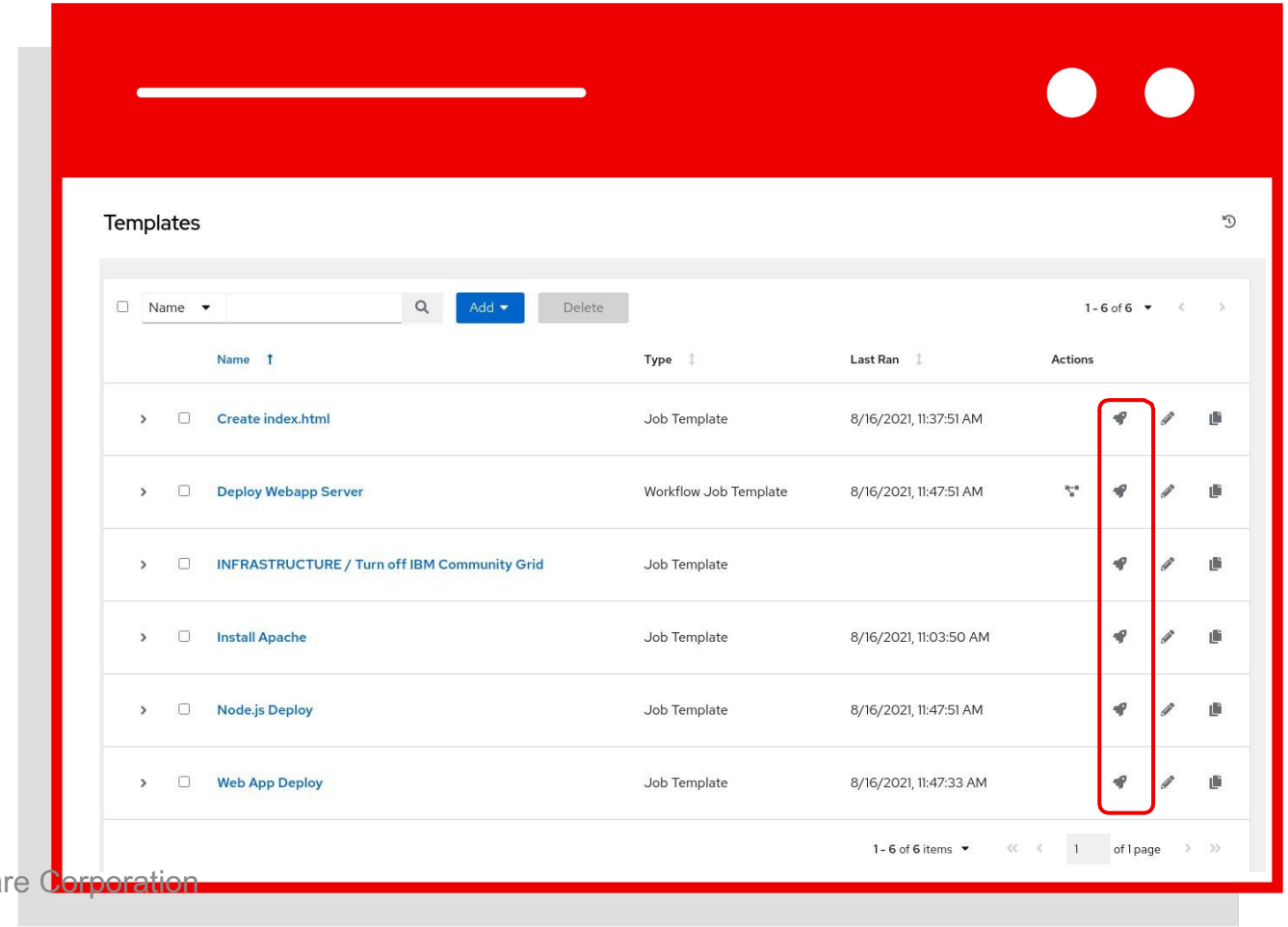
EXPANDING JOB TEMPLATES

Job Templates can be found and created by clicking the **Templates** button under the *Resources* section on the left menu.



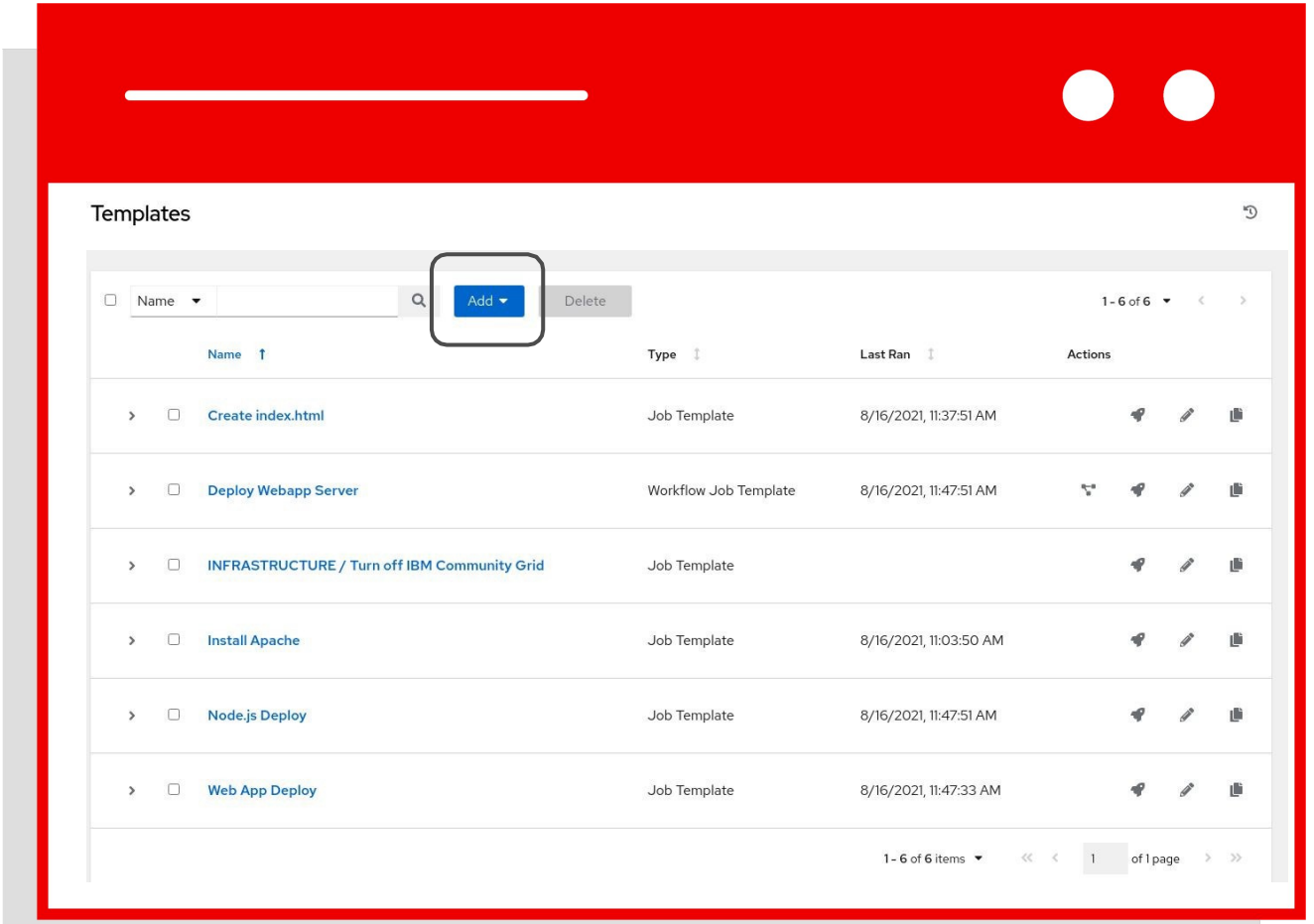
EXECUTING AN EXISTING JOB

Job Templates can be launched by clicking the **rocketship button** for the corresponding Job Template



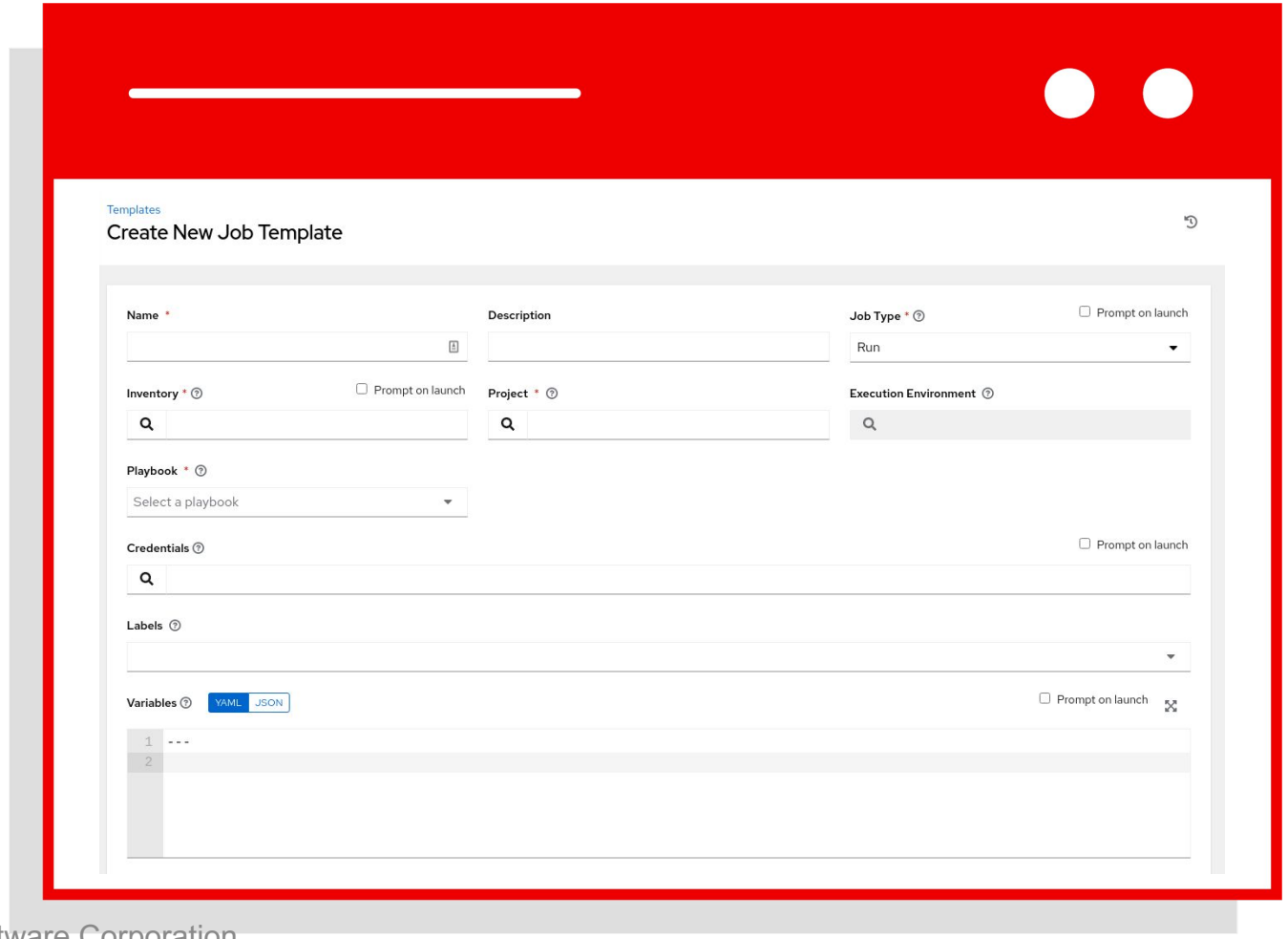
CREATING A NEW JOB TEMPLATE (1/2)

New Job Templates can be created by clicking the **Add button**



CREATING A NEW JOB TEMPLATE (2/2)

This **New Job Template** window is where the inventory, project and credential are assigned. The red asterisk * means the field is required .

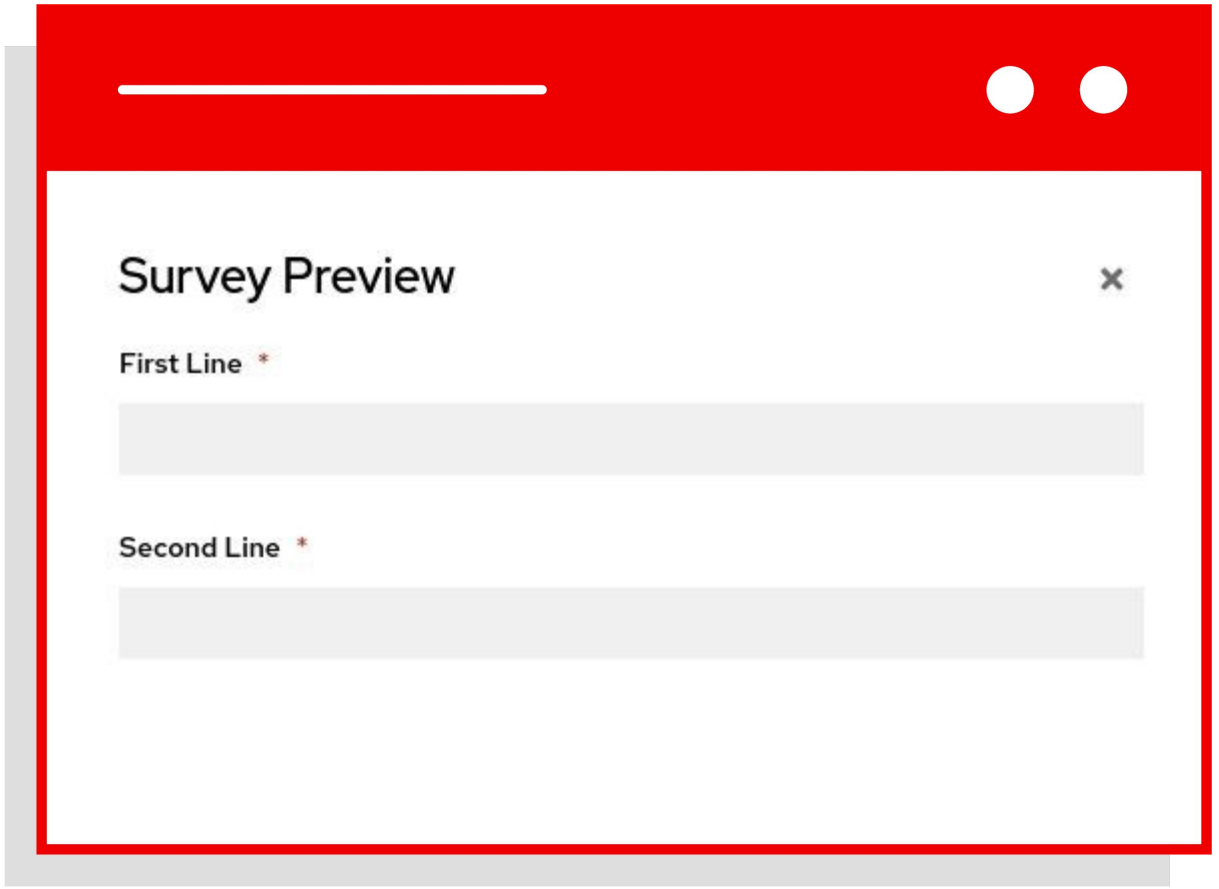


SURVEYS

Controller surveys allow you to configure how a job runs via a series of questions, making it simple to customize your jobs in a user-friendly way.

An Ansible Controller survey is a simple question-and-answer form that allows users to customize their job runs.

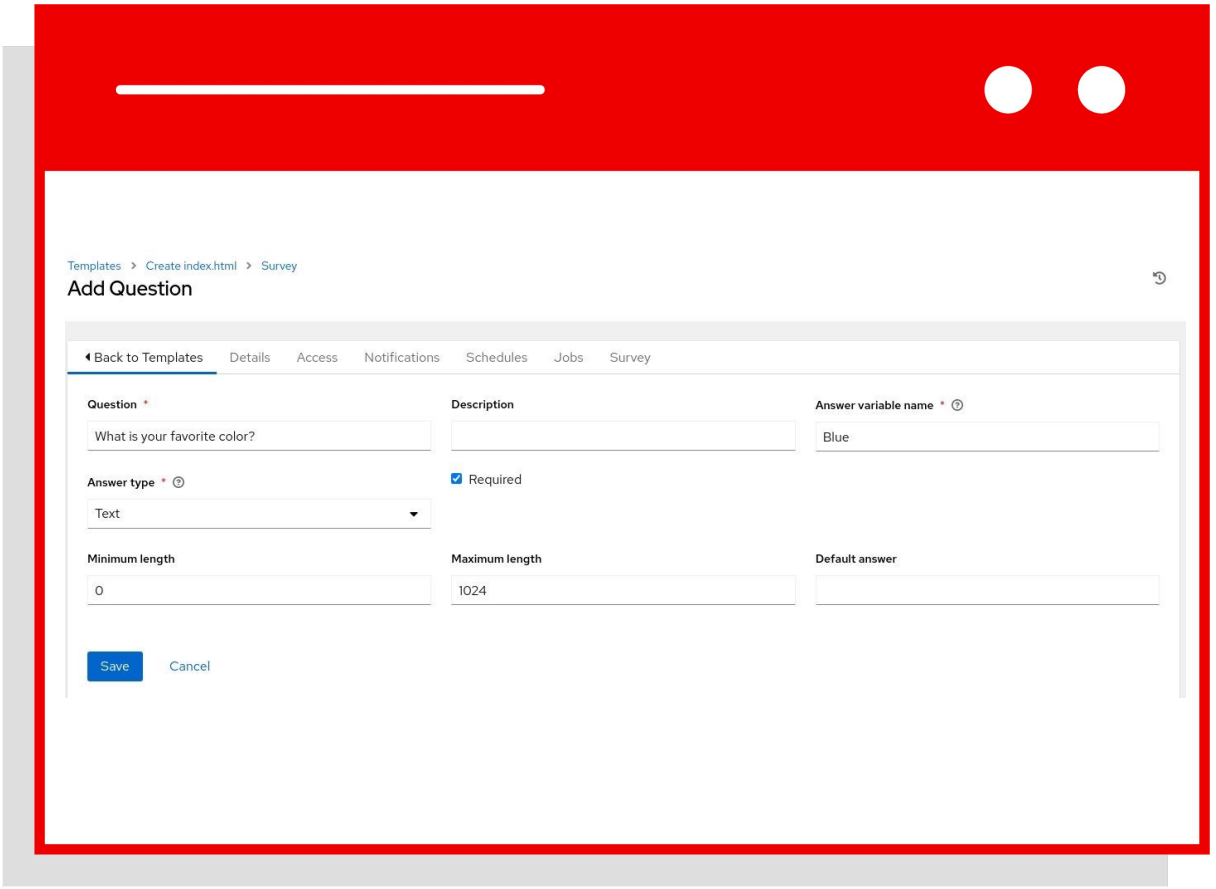
Combine that with Controller's role-based access control, and you can build simple, easy self-service for your users.



CREATING A SURVEY (1/2)

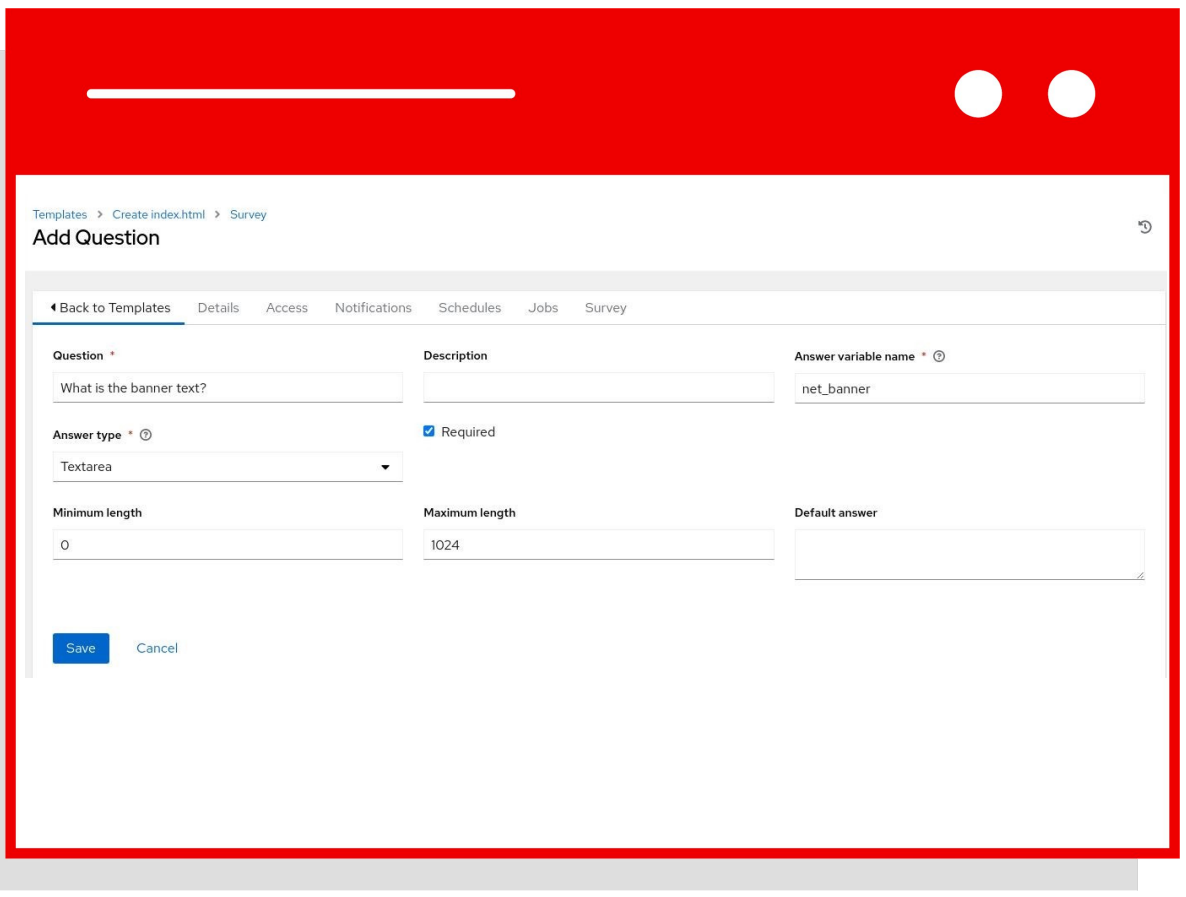
Once a Job Template is saved, the Survey menu will have an **Add Button**

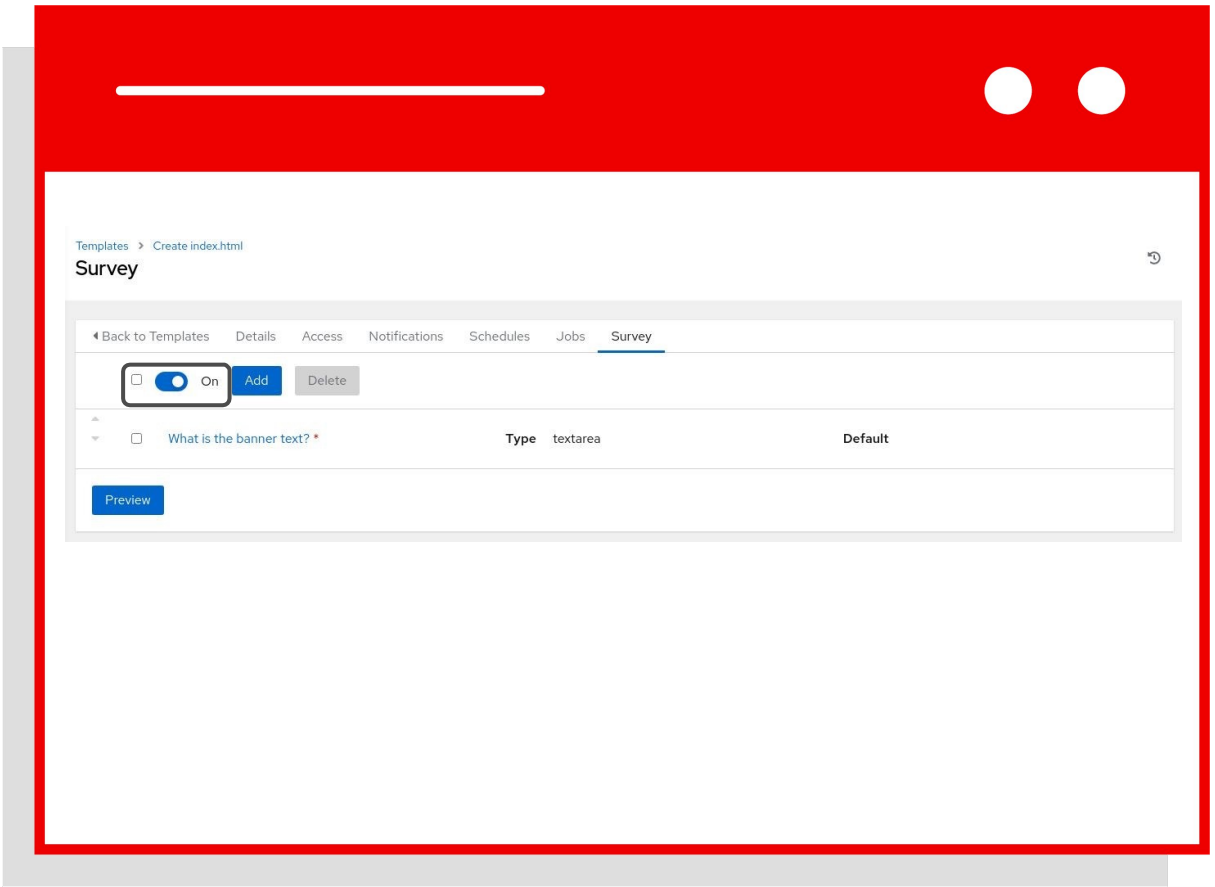
Click the button to open the Add Survey window.



CREATING A SURVEY (2/2)

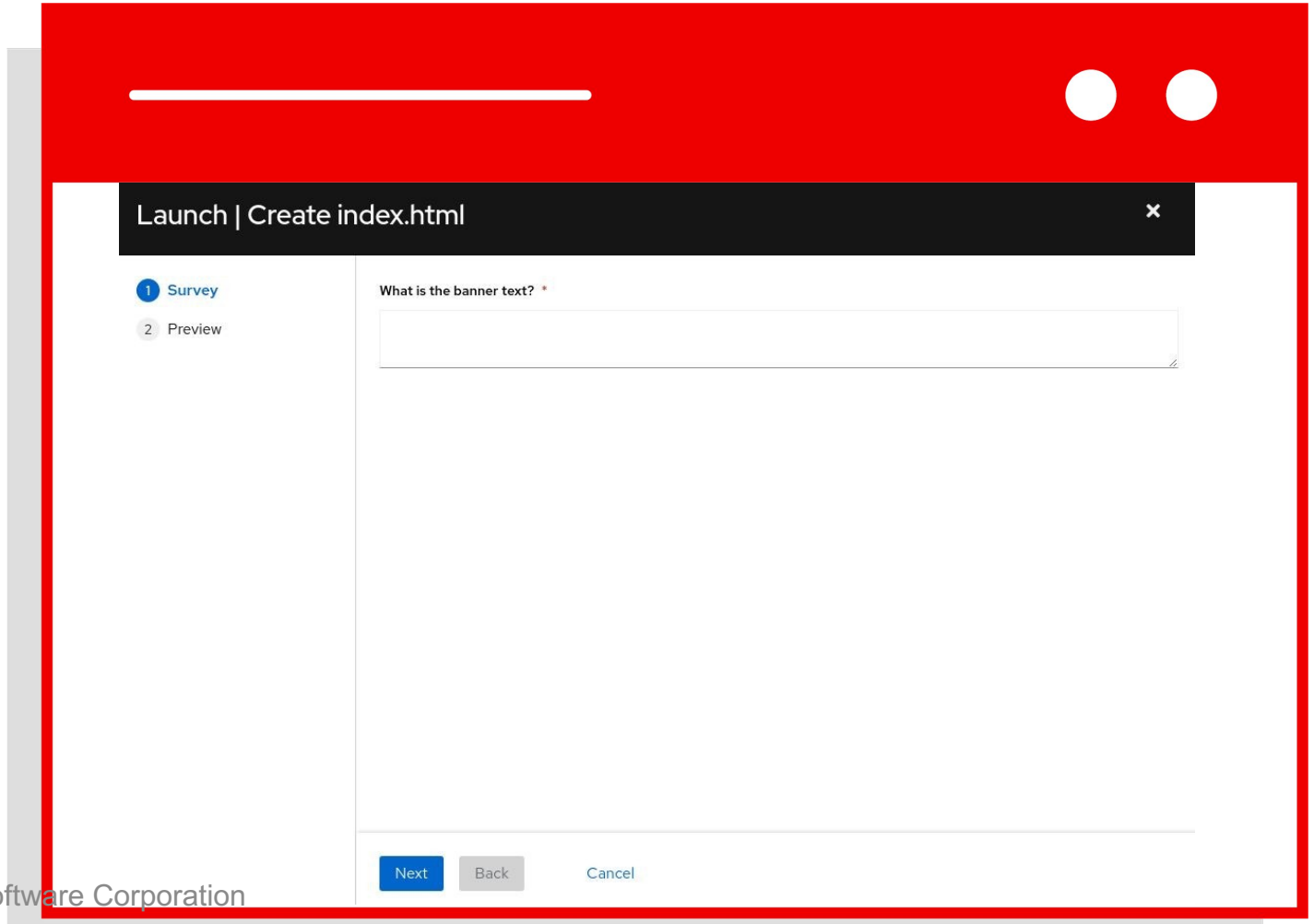
The Add Survey window allows the Job Template to prompt users for one or more questions. The answers provided become variables for use in the Ansible Playbook.





USING A SURVEY

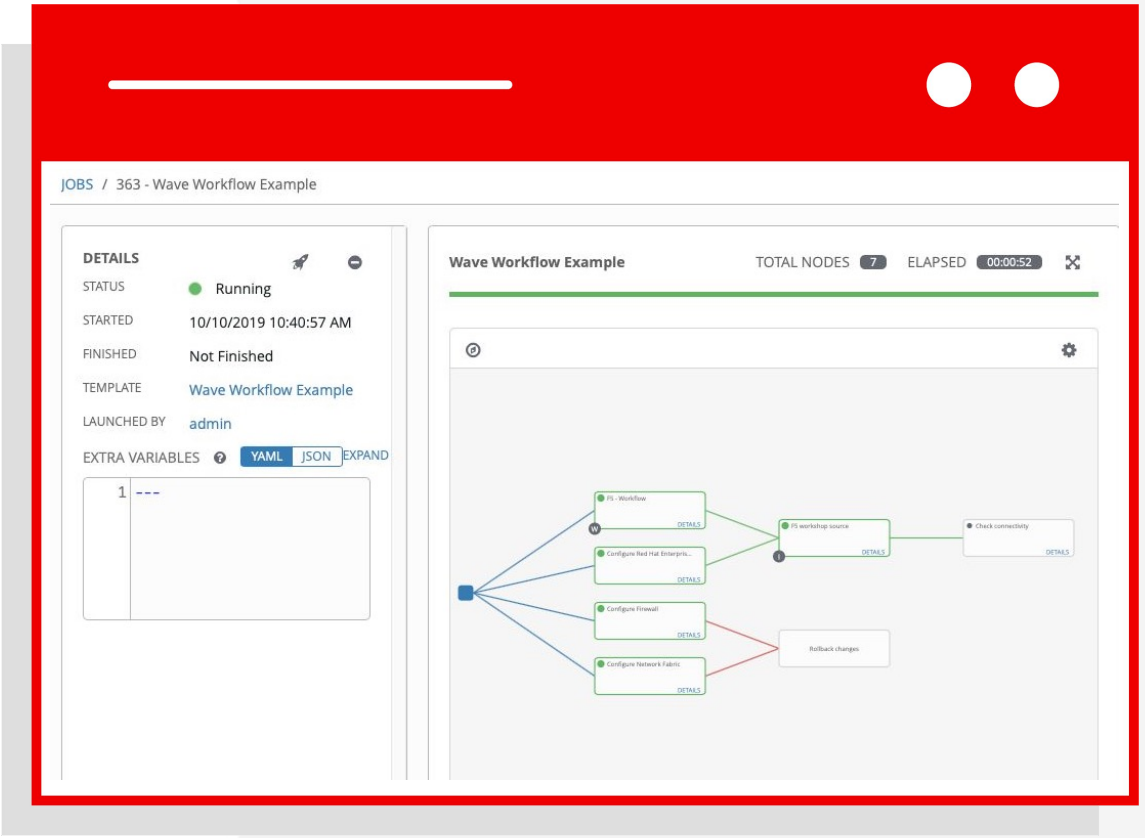
When launching a job, the user will now be prompted with the Survey. The user can be required to fill out the Survey before the Job Template will execute.



WORKFLOWS

Combine automation to create something bigger

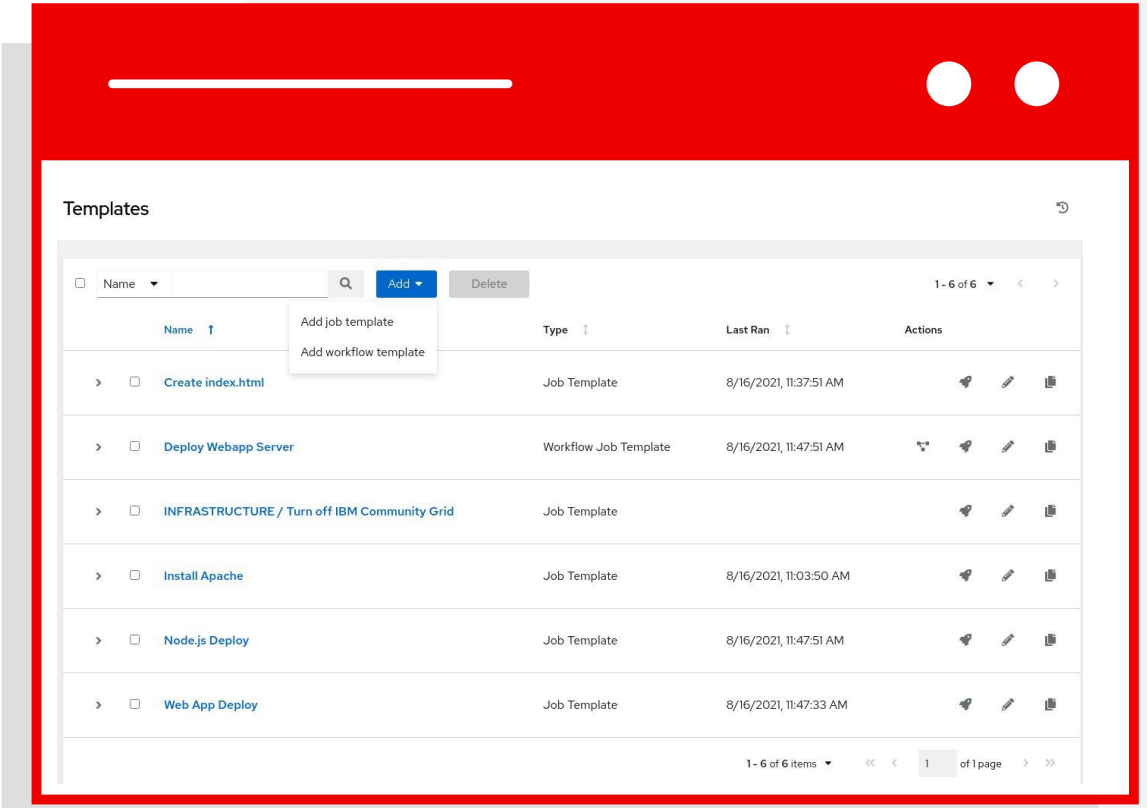
- Workflows enable the creation of powerful holistic automation, chaining together multiple pieces of automation and events.
- Simple logic inside these workflows can trigger automation depending on the success or failure of previous steps.



ADDING A NEW TEMPLATE

- To add a new **Workflow** click on the **Add** button.

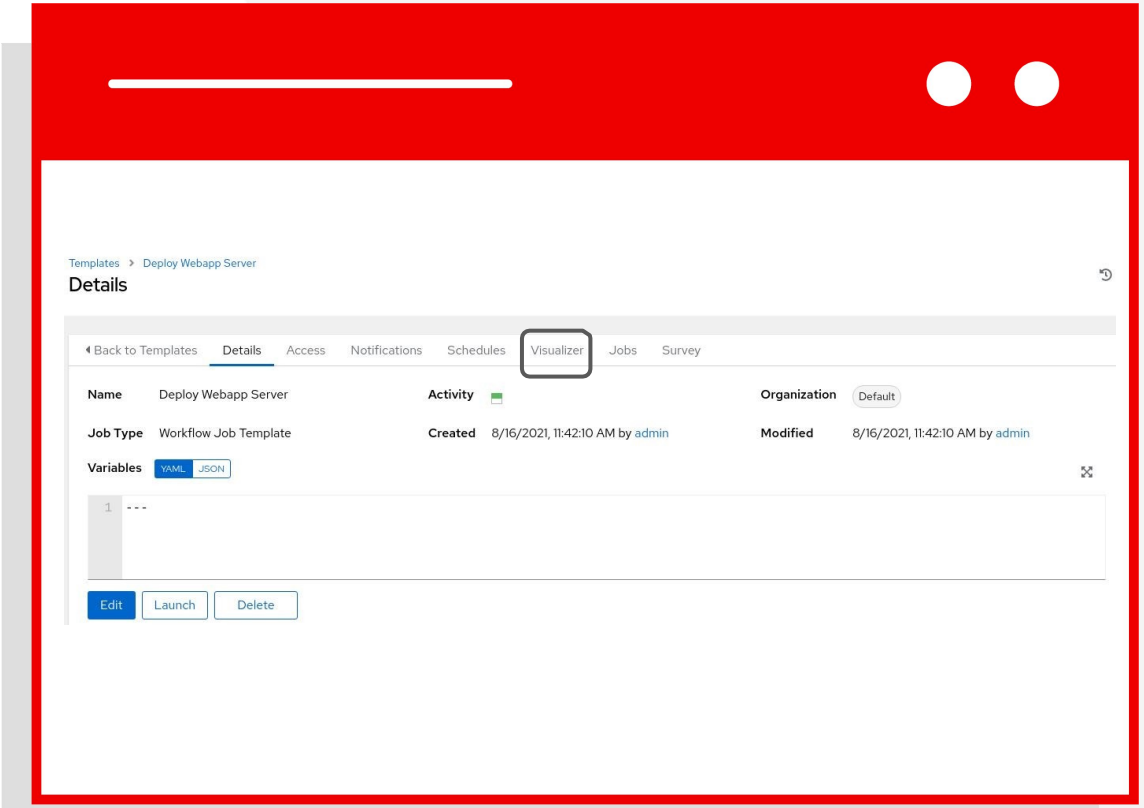
  This time select the **Add workflow template**



CREATING THE WORKFLOW

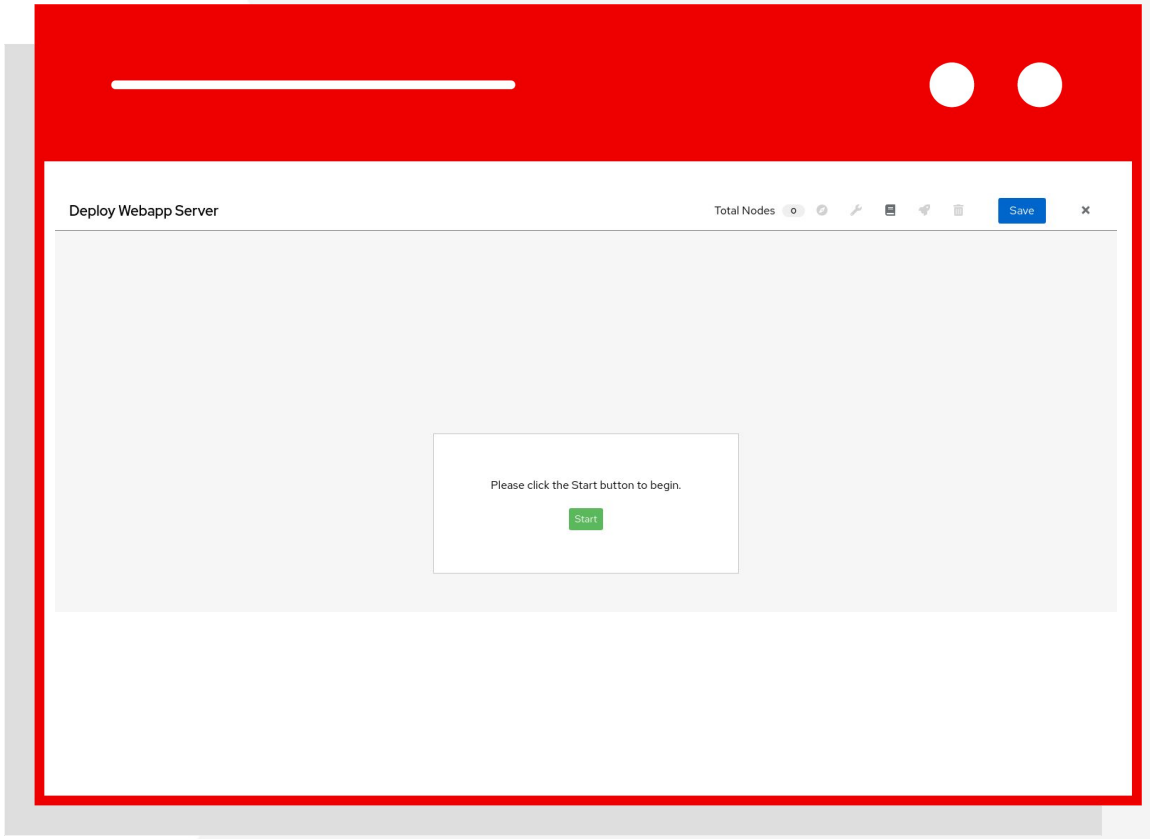
- Fill out the required parameters and click **Save.**

  As soon as the Workflow Template is saved the Workflow Visualizer will open.



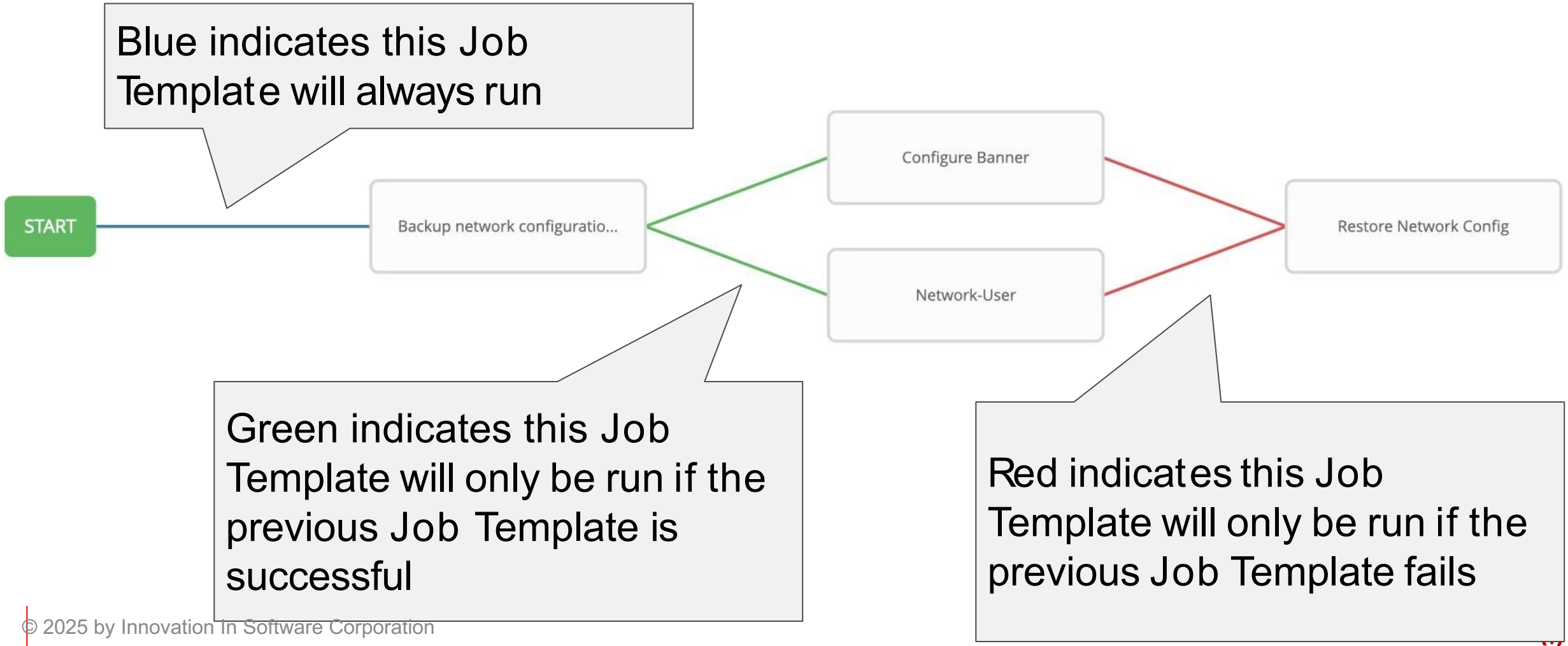
WORKFLOW VISUALIZER

- The Workflow Visualizer will start as a blank canvas.
- Click the green Start button to start building the workflow.



VISUALIZING A WORKFLOW

Workflows can branch out, or converge in.



AAP – Projects and Job Templates

Lab five

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 Practical Example

Practical example (Install nginx and capture failed task)

block_rescue.yml

```yaml
- block:
      - name: Install Nginx
        ansible.builtin.yum:
          name: nginx
          state: present

      - name: Start and enable Nginx service
        ansible.builtin.service:
          name: nginx
          state: started
          enabled: true

    rescue: # These tasks run only if there is a failure on the block
      - name: Show which task failed
        ansible.builtin.debug:
          msg: "{{ ansible_failed_task }}"
```

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.

Force Block Failure

You can force a block to fail even if the rescue task is successful:

fail_block.yaml

```yaml
- block:
    - name: Task 1
      ansible.builtin.debug:
        msg: "A task"

    - name: Task 2
      ansible.builtin.debug:
        cmd: "A second task"

  rescue:
  - name: A rescue task
    ansible.builtin.debug:
      msg: "Rescued"

  - name: Fail the block when rescued
    ansible.builtin.fail:
      msg: "A task failed, so failing the whole block"
```

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'
```

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
```

Ansible Templates