RED HAT® CONSULTING

Git Workflow for Ansible Development

Sam Doran, Senior Software Developer, Ansible by Red Hat

Table of Contents

Git Workflow for Ansible Development

Background

Git is a powerful and flexible distributed version control system. It enables fast and flexible branching and merging. But this increased power and flexibility come at the cost of simplicity, placing the burden on the organization or team to come up with a framework to successfully develop in collaboration.

Ansible does not require the use of a version control system, but storing your Ansible content in a version control repository and adopting traditional software development processes and disciplines is a necessary part of creating scalable automation. Scalable in the sense that it can be shared among a team, not only be of use to the original author, and that it can progress through testing and evaluation environments prior to being run against production systems.

Detailed here is one method recommended to develop Ansible roles and playbooks based on GitLab Flow. It is highly recommended that you read the original documentation when you have additional questions about the workflow. Also, keep in mind that this method is by no means set in stone — feel free to adapt the conventions to meet the needs of your particular team. The important point is everyone working together come to a consensus on the workflow and stick to it.

Playbook Repository Structure

There are many ways to organize Ansible playbooks since Ansible is a very flexible tool. Here is a recommended playbook repository structure that will work consistently on the command line as well as Ansible Tower.

```
playbooks/
├──── group_vars
│     ├──── all.yml
│     ├──── dev.yml
│     ├──── prod.yml
│     └──── web.yml
├──── inventory
├──── library
├──── roles
│     └──── requirements.yml
├──── .gitignore
├──── ansible.cfg
├──── apache.yml
├──── deploy-app.yml
├──── install-updates.yml
└──── site.yml
```

This layout has an `ansible.cfg` at the root of the repository, allowing you to keep in version control the basic settings that are in place when playbooks are run.

Adjacent to playbooks is the group_vars folder. The location of this folder as it relates to the playbooks make these *playbook* group_vars. This is important because these will work when running jobs within Ansible Tower, unlike *inventory* group_vars, which Ansible Tower ignores.

There are several strategies to organizing playbook repositories. The most common is to have one playbook per team, such as the Linux team or Windows team. You may also choose to create playbook repositories based on application. Either approach is fine so long as you maintain the few key aspects of the repository layout detailed here.

One key element of this repository structure is the absence of roles in the playbook repository. The `.gitignore` in this example ignores everything inside the roles directory except for `requirements.yml`:

```
# Ignore everything in roles/ except requirements.yml
roles/*
!roles/requirements.yml
```

We will discuss the advantages to having roles as separate repositories in the next section.

Ansible Tower looks for roles/requirements.yml explicitly when updating projects. If this file is found at this location, the roles listed in the requirements file will be installed. This allows each playbook repository to have its own collection of roles separate from the playbooks. This also allows for easy sharing of roles between playbook repositories.

Roles

Ansible roles provide a convenient way to organize and package Ansible tasks and supporting assets. Organizing Ansible content into roles allows easy reuse across playbooks or playbook repositories. Roles *can* be part of the playbook repository, but creating separate repositories for each role has a number of advantages.

The biggest advantage to creating one repository per role is this allows roles to be shared, either on Ansible Galaxy or across playbook repositories inside your organization.

Having roles reside in separate repositories also allows you to perform unit testing on each role. Testing Ansible roles in a continuous fashion will greatly increase the reliability of your code. With roles in separate repositories, you are also able to tag role releases, allowing strict versioning of individual roles.

Testing Roles

A basic framework for testing Ansible roles is:

- Verify correct syntax by running with `--syntax-check`
- Run the role and ensure it completes with no failures
- Run the role again and ensure no changes are reported
- Query an API or port to ensure the application is functioning

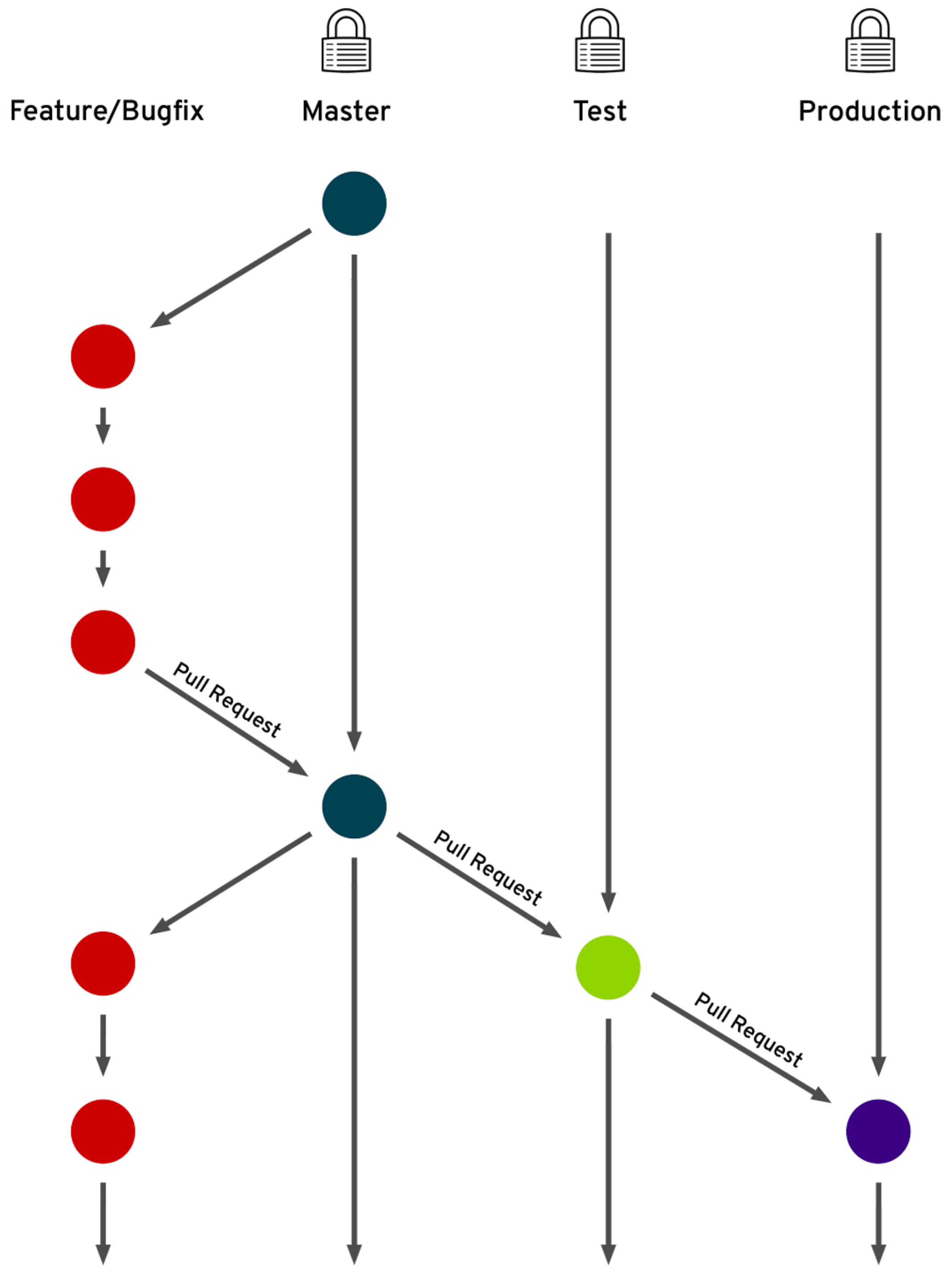
When testing roles, containers provide a convenient way to test your role across multiple distributions quickly. Containers are fine for testing unless you doing low level system things such as modifying the firewall or boot loader settings. In those cases, using virtual machines is recommended.

Numerous CI tools exist for testing code. The exact configuration will vary, but here is an example of testing an Ansible role against multiple Linux distributions inside containers using Travis CI:

```
services: docker

env:
  - distro: centos7
    init: /usr/lib/systemd/systemd
    run_opts: "--privileged --volume=/sys/fs/cgroup:/sys/fs/cgroup:ro"
  - distro: ubuntu16
    init: /lib/systemd/systemd
    run_opts: "--privileged --volume=/sys/fs/cgroup:/sys/fs/cgroup:ro"

before_install:
  # Pull container.
  - 'docker pull samdoran/${distro}-ansible:latest'

script:
  - container_id=$(date +%s)

  # Run container in detached state.
  - echo "Running container"
  - docker run --detach --volume="$PWD":/etc/ansible/roles/role_under_test:ro
--name $container_id $run_opts samdoran/${distro}-ansible:latest $init

  # Install dependencies
  - echo "Installing role dependencies"
  - docker exec --tty $container_id ansible-galaxy install -r
/etc/ansible/roles/role_under_test/tests/requirements.yml

  # Ansible syntax check.
  - echo "Checking role syntax"
  - docker exec --tty $container_id ansible-playbook
/etc/ansible/roles/role_under_test/tests/travis.yml --syntax-check

  # Test role.
  - echo "Running the role"
  - docker exec --tty $container_id ansible-playbook
/etc/ansible/roles/role_under_test/tests/travis.yml

  # Test role idempotence.
  - echo "Testing role idempotence"
  - idempotence_log=$(mktemp)
  - docker exec --tty $container_id ansible-playbook
/etc/ansible/roles/role_under_test/tests/travis.yml | tee -a $idempotence_log
  - >
    tail $idempotence_log
    | grep -q 'changed=0.*failed=0'
    && (echo 'Idempotence test: pass' && exit 0)
    || (echo 'Idempotence test: fail' && exit 1)

notifications:
  webhooks: https://galaxy.ansible.com/api/v1/notifications/
```

This configuration will run the role inside a CentOS 7 container then an Ubuntu 16 container, verify the role completes successfully, then verify subsequent runs of the role do not report any changes. More environments can easily be added the test by building additional containers and adding them to the `env` section.

Ansible Role and Playbook Branching Model



Branches

Branches in git allow you to create an isolated area to work on a specific feature or bug. They should be topical and short-lived. Branches that exist for longer than two weeks, on average, are considered orphaned branches — they have fallen behind the master branch and their commits may no longer merge cleanly. These branches should be rebased and merged, or deleted. Orphaned branches appearing often is a sign of assigning units of work that are too large. Bite off smaller chunks of work, merge early, merge often.

Branching with GitLab flow involve a default unstable branch, usually `master`, from which all feature and bug fix branches originate. The unstable, default branch is used in the development environment, and all feature and bug fixes begin by branching off of this default branch. This is what is commonly called "upstream first" development.

There are also corresponding environment branches for testing and production (or as many environments as you desire, but try to keep it to a small number). All of these branches are protected, meaning no one can commit directly to them — all merges to the protected branches happen through pull requests in GitHub (or equivalent git platform), requiring approval from one or more individuals who did not create the pull request. This creates accountability and prevents unauthorized code from making its way into critical environments.

Pull Requests

Pull requests are more than about merging code — they also provide a forum for discussing code as it is developed to get input on new features.

In the most common used of pull requests, when work in a feature branch is complete, submit a pull request against the `master` branch and assign that pull request to someone who has merge access to the branch. The assignee will then perform a code review and approve the merge (unless more than one approver is required by your organization) or make suggestions and require changes. Any new commits to address issues brought up in review are made to the feature branch and they will automatically show up in the pull request.

Pull requests can also be used as a forum for discussion. To signal the pull request is not ready for merge, a common convention is to prefix the pull request title with "WIP:" and not assign it to anyone. In the comments, @mention team mates you would like to invite to look at the code.

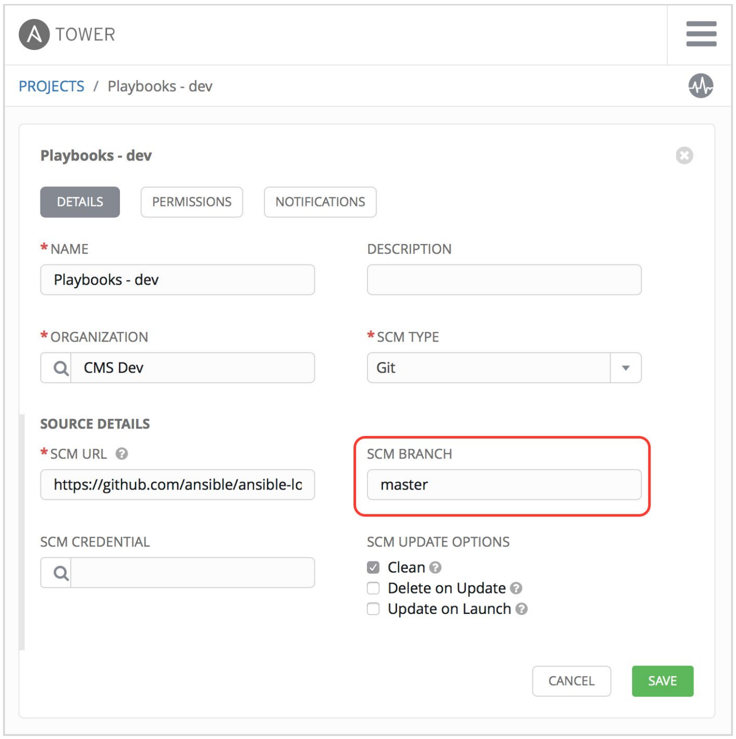
Once that feature is complete and ready to be merged, remove the "WIP:" prefix from the title and assign it to someone with merge access to the `master` branch. From there, the normal code review and merge process takes place.

Ansible Tower

Tying this branching model together with Ansible Tower is the final step.

Projects

A Project in Ansible Tower is a repository and a branch. Create one project per environment branch and ensure that only a small number of people have the ability to edit this Project. Since these projects will be tied to Job Templates which will run and make changes to hosts, it's important that the Project can only be modified by authorized persons.



TOWER

PROJECTS / Playbooks - dev

Playbooks - dev

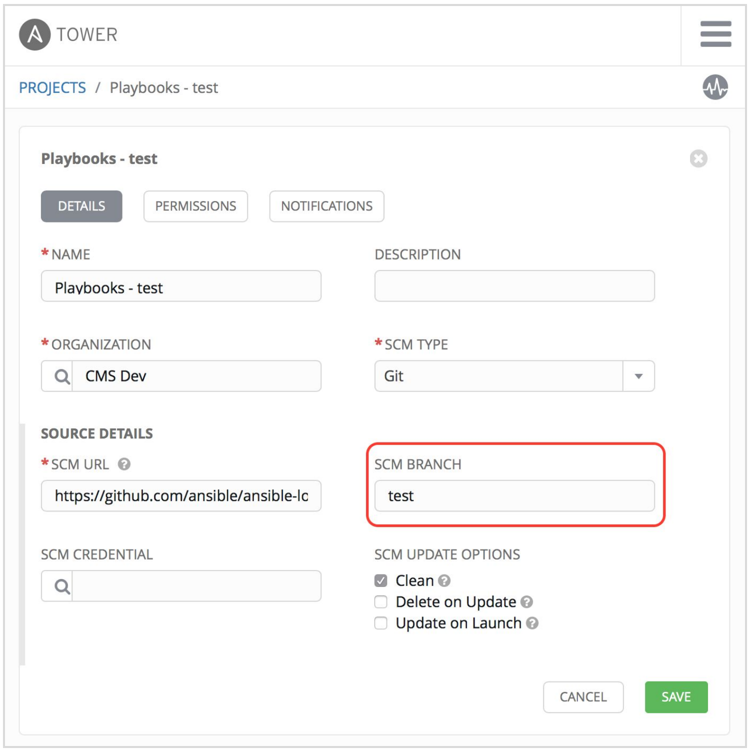
DETAILS | PERMISSIONS | NOTIFICATIONS

* NAME: Playbooks - dev	DESCRIPTION:
* ORGANIZATION: CMS Dev	* SCM TYPE: Git

SOURCE DETAILS

* SCM URL: https://github.com/ansible/ansible-lc	SCM BRANCH: master
SCM CREDENTIAL:	SCM UPDATE OPTIONS: [✓] Clean; [ ] Delete on Update; [ ] Update on Launch

CANCEL SAVE



TOWER

PROJECTS / Playbooks - test

Playbooks - test

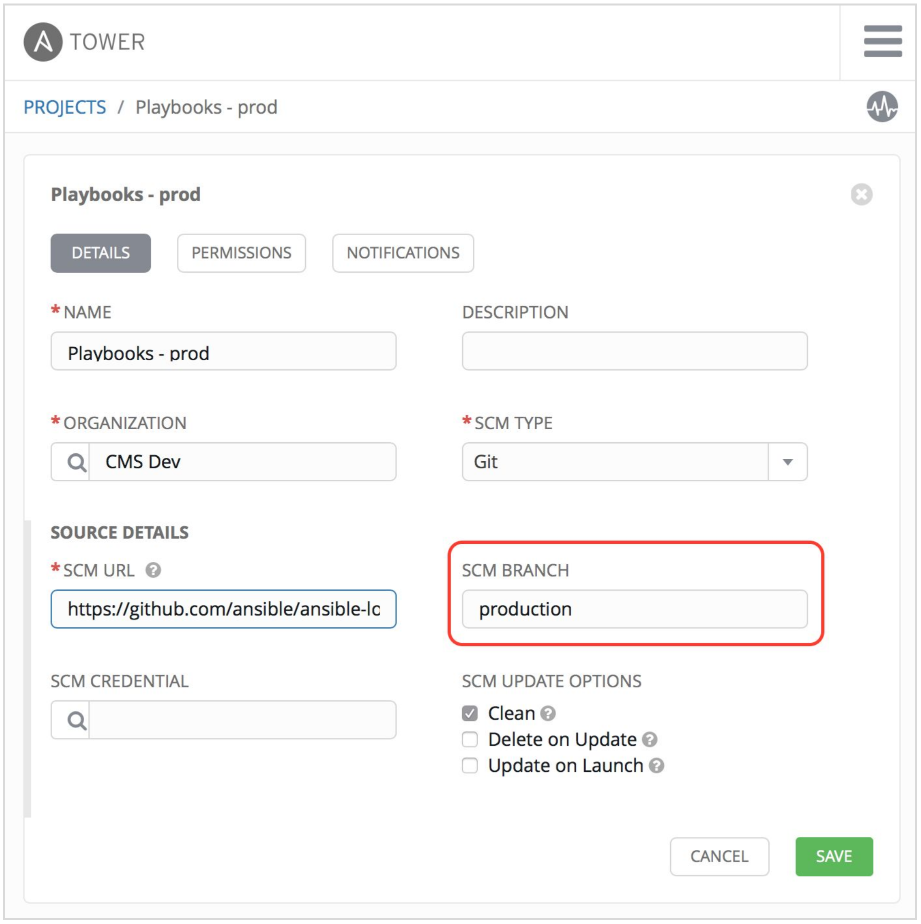
DETAILS | PERMISSIONS | NOTIFICATIONS

* NAME: Playbooks - test	DESCRIPTION:
* ORGANIZATION: CMS Dev	* SCM TYPE: Git

SOURCE DETAILS

* SCM URL: https://github.com/ansible/ansible-lc	SCM BRANCH: test
SCM CREDENTIAL:	SCM UPDATE OPTIONS: [✓] Clean; [ ] Delete on Update; [ ] Update on Launch

CANCEL SAVE


TOWER

PROJECTS / Playbooks - prod

Playbooks - prod

DETAILS | PERMISSIONS | NOTIFICATIONS

* NAME: Playbooks - prod	DESCRIPTION:
* ORGANIZATION: CMS Dev	* SCM TYPE: Git

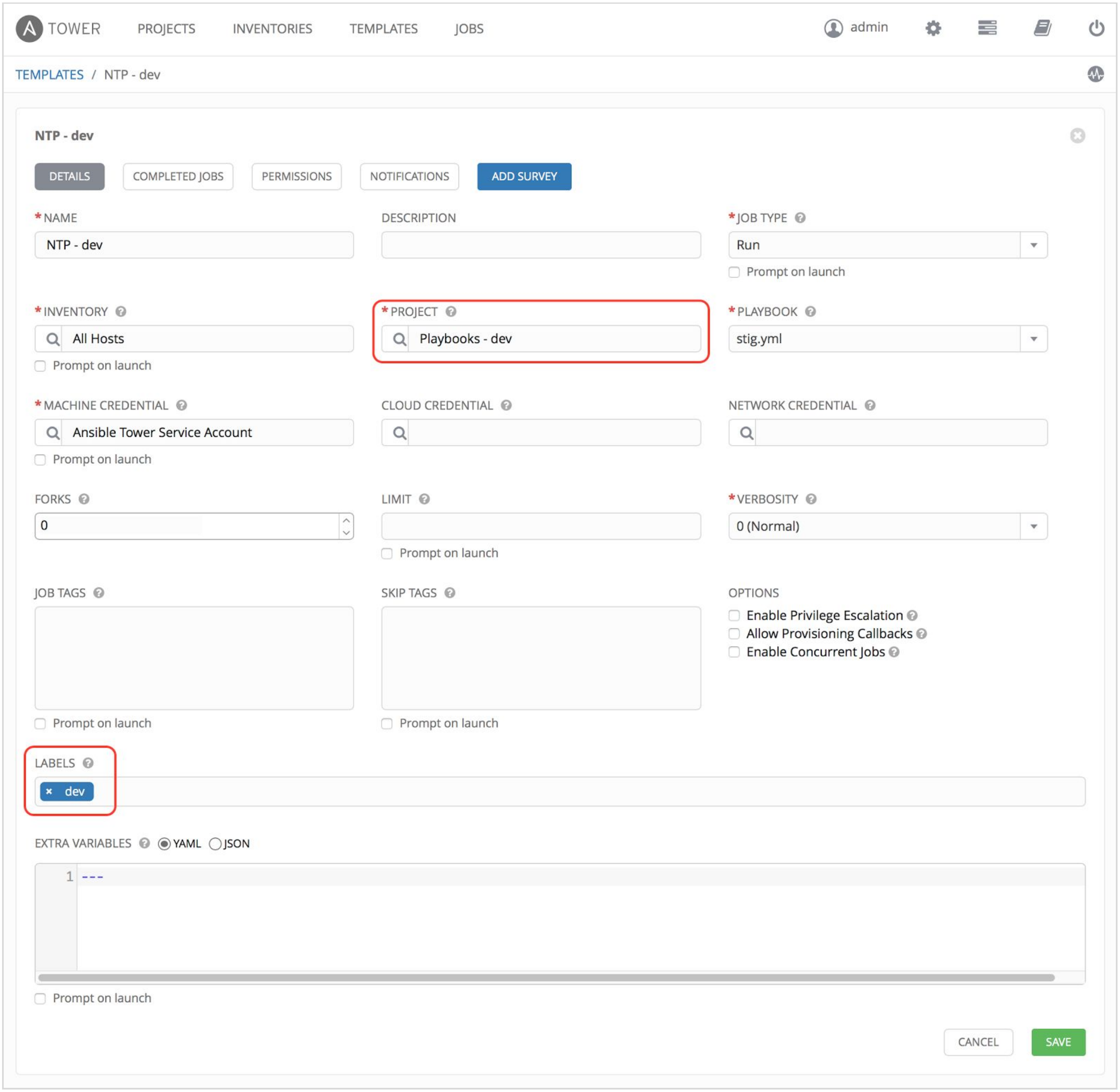
SOURCE DETAILS

* SCM URL: https://github.com/ansible/ansible-lc	SCM BRANCH: production
SCM CREDENTIAL:	SCM UPDATE OPTIONS: [✓] Clean; [ ] Delete on Update; [ ] Update on Launch

CANCEL SAVE

Job Templates

Next, create Job Templates for each environment that point to the appropriate Project (branch). You can use labels in Ansible Tower and the search field to make it easier to organize your Job Templates. It is normal to have a lot of Job Templates.



TEMPLATES / NTP - dev

NTP - dev

*NAME: NTP - dev

DESCRIPTION:

*JOB TYPE: Run

☐ Prompt on launch

*INVENTORY: All Hosts

☐ Prompt on launch

*PROJECT: Playbooks - dev

*PLAYBOOK: stig.yml

*MACHINE CREDENTIAL: Ansible Tower Service Account

☐ Prompt on launch

CLOUD CREDENTIAL:

NETWORK CREDENTIAL:

FORKS: 0

LIMIT:

☐ Prompt on launch

*VERBOSITY: 0 (Normal)

JOB TAGS:

☐ Prompt on launch

SKIP TAGS:

☐ Prompt on launch

OPTIONS

☐ Enable Privilege Escalation

☐ Allow Provisioning Callbacks

☐ Enable Concurrent Jobs

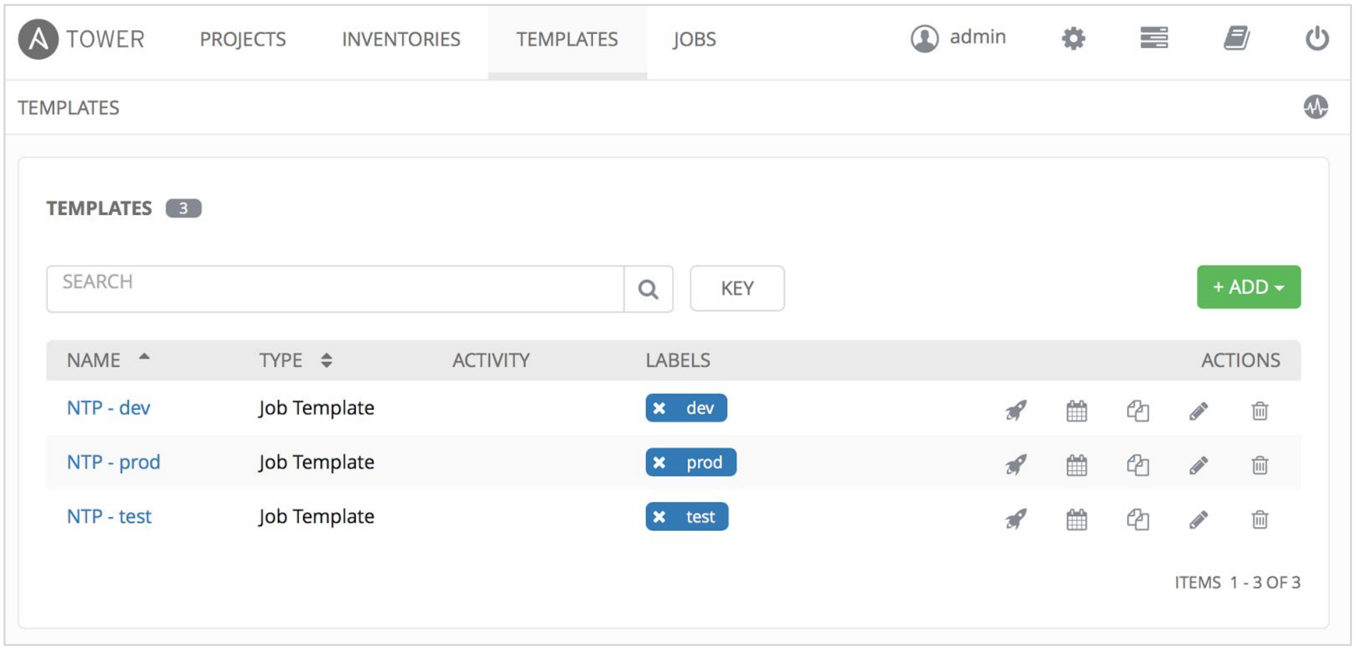
LABELS: dev

EXTRA VARIABLES [✓ YAML] JSON

```
---
```

☐ Prompt on launch

Select the project and add labels

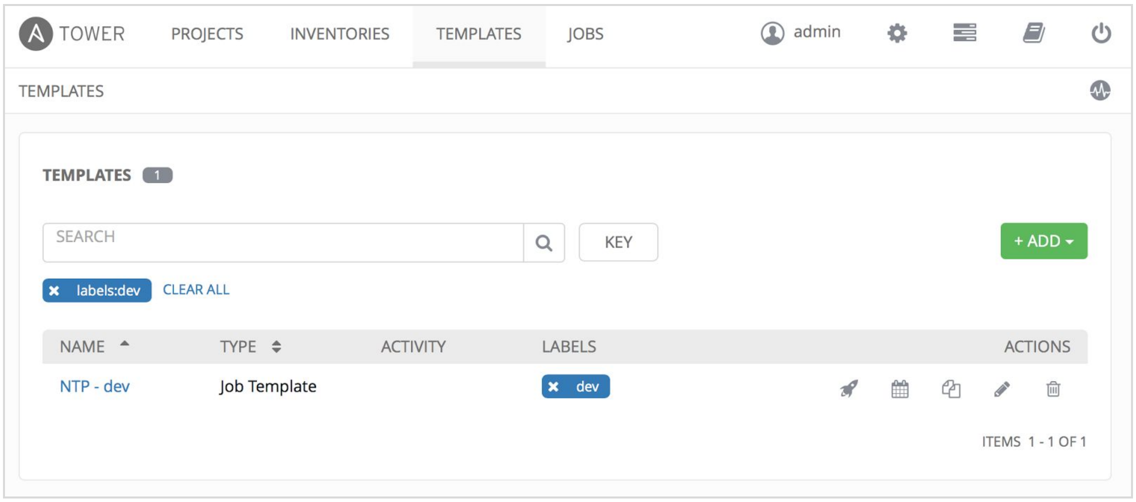


NAME	TYPE	ACTIVITY	LABELS	ACTIONS
NTP - dev	Job Template		dev	
NTP - prod	Job Template		prod	
NTP - test	Job Template		test	

ITEMS 1 - 3 OF 3

Create one Job Template per playbook per environment

To make it easier to view environment specific playbooks, use the search field. For instance, create a search to show all the dev Job Templates by entering `labels:dev` in the search box.



NAME	TYPE	ACTIVITY	LABELS	ACTIONS
NTP - dev	Job Template		dev	

ITEMS 1 - 1 OF 1

These search terms become part of the URL, so you may bookmark them for easy access.