Playbook Example: Continuous Delivery and Rolling Upgrades

What is continuous delivery?

Continuous delivery (CD) means frequently delivering updates to your software application.

The idea is that by updating more often, you do not have to wait for a specific timed period, and your organization gets better at the process of responding to change.

Some Ansible users are deploying updates to their end users on an hourly or even more frequent basis -- sometimes every time there is an approved code change. To achieve this, you need tools to be able to quickly apply those updates in a zero-downtime way.

This document describes in detail how to achieve this goal, using one of Ansible's most complete example playbooks as a template: lamp_haproxy. This example uses a lot of Ansible features: roles, templates, and group variables, and it also comes with an orchestration playbook that can do zero-downtime rolling upgrades of the web application stack.

> **Note**
>
> Click here for the latest playbooks for this example.

The playbooks deploy Apache, PHP, MySQL, Nagios, and HAProxy to a CentOS-based set of servers.

We're not going to cover how to run these playbooks here. Read the included README in the github project along with the example for that information. Instead, we're going to take a close look at every part of the playbook and describe what it does.

Site deployment

Let's start with `site.yml`. This is our site-wide deployment playbook. It can be used to initially deploy the site, as well as push updates to all of the servers:

```
---
# This playbook deploys the whole application stack in this site.

# Apply common configuration to all hosts
- hosts: all

  roles:
  - common

# Configure and deploy database servers.
- hosts: dbservers

  roles:
  - db

# Configure and deploy the web servers. Note that we include two roles
# here, the 'base-apache' role which simply sets up Apache, and 'web'
# which includes our example web application.

- hosts: webservers

  roles:
  - base-apache
  - web

# Configure and deploy the load balancer(s).
- hosts: lbservers

  roles:
  - haproxy

# Configure and deploy the Nagios monitoring node(s).
```

```
- hosts: monitoring

  roles:
  - base-apache
  - nagios
```

Note

If you're not familiar with terms like playbooks and plays, you should review :ref:`working_with_playbooks`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\user_guide\(ansible-devel)(docs)(docsite)(rst)(user_guide)guide_rolling_upgrade.rst, line 86); *backlink*

Unknown interpreted text role "ref".

In this playbook we have 5 plays. The first one targets `all` hosts and applies the `common` role to all of the hosts. This is for site-wide things like yum repository configuration, firewall configuration, and anything else that needs to apply to all of the servers.

The next four plays run against specific host groups and apply specific roles to those servers. Along with the roles for Nagios monitoring, the database, and the web application, we've implemented a `base-apache` role that installs and configures a basic Apache setup. This is used by both the sample web application and the Nagios hosts.

Reusable content: roles

By now you should have a bit of understanding about roles and how they work in Ansible. Roles are a way to organize content: tasks, handlers, templates, and files, into reusable components.

This example has six roles: `common`, `base-apache`, `db`, `haproxy`, `nagios`, and `web`. How you organize your roles is up to you and your application, but most sites will have one or more common roles that are applied to all systems, and then a series of application-specific roles that install and configure particular parts of the site.

Roles can have variables and dependencies, and you can pass in parameters to roles to modify their behavior. You can read more about roles in the :ref:`playbooks_reuse_roles` section.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\user_guide\(ansible-devel)(docs)(docsite)(rst)(user_guide)guide_rolling_upgrade.rst, line 108); *backlink*

Unknown interpreted text role "ref".

Configuration: group variables

Group variables are variables that are applied to groups of servers. They can be used in templates and in playbooks to customize behavior and to provide easily-changed settings and parameters. They are stored in a directory called `group_vars` in the same location as your inventory. Here is lamp_haproxy's `group_vars/all` file. As you might expect, these variables are applied to all of the machines in your inventory:

```
---
httpd_port: 80
ntpserver: 192.0.2.23
```

This is a YAML file, and you can create lists and dictionaries for more complex variable structures. In this case, we are just setting two variables, one for the port for the web server, and one for the NTP server that our machines should use for time synchronization.

Here's another group variables file. This is `group_vars/dbservers` which applies to the hosts in the `dbservers` group:

```
---
mysqlservice: mysqld
mysql_port: 3306
dbuser: root
dbname: foodb
upassword: usersecret
```

If you look in the example, there are group variables for the `webservers` group and the `lbservers` group, similarly.

These variables are used in a variety of places. You can use them in playbooks, like this, in `roles/db/tasks/main.yml`:

```
- name: Create Application Database
  mysql_db:
    name: "{{ dbname }}"
    state: present
```

```
- name: Create Application DB User
  mysql_user:
    name: "{{ dbuser }}"
    password: "{{ upassword }}"
    priv: "*.*:ALL"
    host: '%'
    state: present
```

You can also use these variables in templates, like this, in `roles/common/templates/ntp.conf.j2`:

```
driftfile /var/lib/ntp/drift

restrict 127.0.0.1
restrict -6 ::1

server {{ ntpserver }}

includefile /etc/ntp/crypto/pw

keys /etc/ntp/keys
```

You can see that the variable substitution syntax of {{ and }} is the same for both templates and variables. The syntax inside the curly braces is Jinja2, and you can do all sorts of operations and apply different filters to the data inside. In templates, you can also use for loops and if statements to handle more complex situations, like this, in `roles/common/templates/iptables.j2`:

```
{% if inventory_hostname in groups['dbservers'] %}
-A INPUT -p tcp  --dport 3306 -j  ACCEPT
{% endif %}
```

This is testing to see if the inventory name of the machine we're currently operating on (`inventory_hostname`) exists in the inventory group `dbservers`. If so, that machine will get an iptables ACCEPT line for port 3306.

Here's another example, from the same template:

```
{% for host in groups['monitoring'] %}
-A INPUT -p tcp -s {{ hostvars[host].ansible_default_ipv4.address }} --dport 5666 -j ACCEPT
{% endfor %}
```

This loops over all of the hosts in the group called `monitoring`, and adds an ACCEPT line for each monitoring hosts' default IPv4 address to the current machine's iptables configuration, so that Nagios can monitor those hosts.

You can learn a lot more about Jinja2 and its capabilities here, and you can read more about Ansible variables in general in the :ref:`playbooks_variables` section.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\user_guide\(ansible-devel)(docs)(docsite)(rst)(user_guide)guide_rolling_upgrade.rst`, line 201); *backlink*
>
> Unknown interpreted text role "ref".

The rolling upgrade

Now you have a fully-deployed site with web servers, a load balancer, and monitoring. How do you update it? This is where Ansible's orchestration features come into play. While some applications use the term 'orchestration' to mean basic ordering or command-blasting, Ansible refers to orchestration as 'conducting machines like an orchestra', and has a pretty sophisticated engine for it.

Ansible has the capability to do operations on multi-tier applications in a coordinated way, making it easy to orchestrate a sophisticated zero-downtime rolling upgrade of our web application. This is implemented in a separate playbook, called `rolling_update.yml`.

Looking at the playbook, you can see it is made up of two plays. The first play is very simple and looks like this:

```
- hosts: monitoring
  tasks: []
```

What's going on here, and why are there no tasks? You might know that Ansible gathers "facts" from the servers before operating upon them. These facts are useful for all sorts of things: networking information, OS/distribution versions, and so on. In our case, we need to know something about all of the monitoring servers in our environment before we perform the update, so this simple play forces a fact-gathering step on our monitoring servers. You will see this pattern sometimes, and it's a useful trick to know.

The next part is the update play. The first part looks like this:

```
- hosts: webservers
  user: root
```

```
  serial: 1
```

This is just a normal play definition, operating on the `webservers` group. The `serial` keyword tells Ansible how many servers to operate on at once. If it's not specified, Ansible will parallelize these operations up to the default "forks" limit specified in the configuration file. But for a zero-downtime rolling upgrade, you may not want to operate on that many hosts at once. If you had just a handful of webservers, you may want to set `serial` to 1, for one host at a time. If you have 100, maybe you could set `serial` to 10, for ten at a time.

Here is the next part of the update play:

```
pre_tasks:
- name: disable nagios alerts for this host webserver service
  nagios:
    action: disable_alerts
    host: "{{ inventory_hostname }}"
    services: webserver
  delegate_to: "{{ item }}"
  loop: "{{ groups.monitoring }}"

- name: disable the server in haproxy
  shell: echo "disable server myapplb/{{ inventory_hostname }}" | socat stdio /var/lib/haproxy/stats
  delegate_to: "{{ item }}"
  loop: "{{ groups.lbservers }}"
```

> **Note**
>
> - The `serial` keyword forces the play to be executed in 'batches'. Each batch counts as a full play with a subselection of hosts. This has some consequences on play behavior. For example, if all hosts in a batch fails, the play fails, which in turn fails the entire run. You should consider this when combining with `max_fail_percentage`.

The `pre_tasks` keyword just lets you list tasks to run before the roles are called. This will make more sense in a minute. If you look at the names of these tasks, you can see that we are disabling Nagios alerts and then removing the webserver that we are currently updating from the HAProxy load balancing pool.

The `delegate_to` and `loop` arguments, used together, cause Ansible to loop over each monitoring server and load balancer, and perform that operation (delegate that operation) on the monitoring or load balancing server, "on behalf" of the webserver. In programming terms, the outer loop is the list of web servers, and the inner loop is the list of monitoring servers.

Note that the HAProxy step looks a little complicated. We're using HAProxy in this example because it's freely available, though if you have (for instance) an F5 or Netscaler in your infrastructure (or maybe you have an AWS Elastic IP setup?), you can use Ansible modules to communicate with them instead. You might also wish to use other monitoring modules instead of nagios, but this just shows the main goal of the 'pre tasks' section -- take the server out of monitoring, and take it out of rotation.

The next step simply re-applies the proper roles to the web servers. This will cause any configuration management declarations in `web` and `base-apache` roles to be applied to the web servers, including an update of the web application code itself. We don't have to do it this way--we could instead just purely update the web application, but this is a good example of how roles can be used to reuse tasks:

```
roles:
- common
- base-apache
- web
```

Finally, in the `post_tasks` section, we reverse the changes to the Nagios configuration and put the web server back in the load balancing pool:

```
post_tasks:
- name: Enable the server in haproxy
  shell: echo "enable server myapplb/{{ inventory_hostname }}" | socat stdio /var/lib/haproxy/stats
  delegate_to: "{{ item }}"
  loop: "{{ groups.lbservers }}"

- name: re-enable nagios alerts
  nagios:
    action: enable_alerts
    host: "{{ inventory_hostname }}"
    services: webserver
  delegate_to: "{{ item }}"
  loop: "{{ groups.monitoring }}"
```

Again, if you were using a Netscaler or F5 or Elastic Load Balancer, you would just substitute in the appropriate modules instead.

Managing other load balancers

In this example, we use the simple HAProxy load balancer to front-end the web servers. It's easy to configure and easy to manage. As we have mentioned, Ansible has support for a variety of other load balancers like Citrix NetScaler, F5 BigIP, Amazon Elastic Load Balancers, and more. See the :ref:`working_with_modules` documentation for more information.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\user_guide\(ansible-devel)(docs)(docsite)(rst)(user_guide)guide_rolling_upgrade.rst, line 296); *backlink***
>
> Unknown interpreted text role "ref".

For other load balancers, you may need to send shell commands to them (like we do for HAProxy above), or call an API, if your load balancer exposes one. For the load balancers for which Ansible has modules, you may want to run them as a `local_action` if they contact an API. You can read more about local actions in the :ref:`playbooks_delegation` section. Should you develop anything interesting for some hardware where there is not a module, it might make for a good contribution!

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\user_guide\(ansible-devel)(docs)(docsite)(rst)(user_guide)guide_rolling_upgrade.rst, line 298); *backlink***
>
> Unknown interpreted text role "ref".

Continuous delivery end-to-end

Now that you have an automated way to deploy updates to your application, how do you tie it all together? A lot of organizations use a continuous integration tool like Jenkins or Atlassian Bamboo to tie the development, test, release, and deploy steps together. You may also want to use a tool like Gerrit to add a code review step to commits to either the application code itself, or to your Ansible playbooks, or both.

Depending on your environment, you might be deploying continuously to a test environment, running an integration test battery against that environment, and then deploying automatically into production. Or you could keep it simple and just use the rolling-update for on-demand deployment into test or production specifically. This is all up to you.

For integration with Continuous Integration systems, you can easily trigger playbook runs using the `ansible-playbook` command line tool, or, if you're using AWX, the `tower-cli` command or the built-in REST API. (The tower-cli command 'joblaunch' will spawn a remote job over the REST API and is pretty slick).

This should give you a good idea of how to structure a multi-tier application with Ansible, and orchestrate operations upon that app, with the eventual goal of continuous delivery to your customers. You could extend the idea of the rolling upgrade to lots of different parts of the app; maybe add front-end web servers along with application servers, for instance, or replace the SQL database with something like MongoDB or Riak. Ansible gives you the capability to easily manage complicated environments and automate common operations.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\user_guide\(ansible-devel)(docs)(docsite)(rst)(user_guide)guide_rolling_upgrade.rst, line 313)**
>
> Unknown directive type "seealso".
>
> ```
> .. seealso::
>
>    `lamp_haproxy example <https://github.com/ansible/ansible-examples/tree/master/lamp_haproxy>`_
>        The lamp_haproxy example discussed here.
>    :ref:`working_with_playbooks`
>        An introduction to playbooks
>    :ref:`playbooks_reuse_roles`
>        An introduction to playbook roles
>    :ref:`playbooks_variables`
>        An introduction to Ansible variables
>    `Ansible.com: Continuous Delivery <https://www.ansible.com/use-cases/continuous-delivery>`_
>        An introduction to Continuous Delivery with Ansible
> ```