**Implementation with Ansible**

[Ansible](http://bit.ly/1PrevnF) is an open source software that had been developed to perform automated software provisioning, configuration management and application deployment. In a typical deployment setup, there is a master server (”control machine”) and multiple slaves (”nodes”). The actual control process is running on the master. During its execution, it can run processes on the slaves, transfer files or run processes on the master itself.

To get access to the slaves, the master just needs an ssh account on the target machine. Authentication is usually done with SSH keys. The advantage of SSH is that Ansible can run any command on the slaves, just as if a user was logged in. The account’s privileges can be fine tuned so that only privileges that are required for the intended usage are granted.

**Distributing the Application**

To have a consistent application state, we first collect all files that need to be deployed - application files, templates, resource files etc. - and put them together into a bundle. This bundle is just a simple .tar.gz file.

This bundle now needs to be distributed to the production servers. This is the first task where Ansible can take over. On the Ansible master, a control process is started that executed a **playbook**.

Playbooks are written in [YAML](https://en.wikipedia.org/wiki/YAML), as are most Ansible configuration files. Each playbook consists of a list of *tasks* and operates on a selection of *hosts* (i.e. the slaves). Each task calls a *module* that is executed on each of the hosts.

Typical tasks are:

* copy a file from the master to the slave
* copy/move files or directories between locations on the slave
* extract an archive file
* create or delete a file/directory
* change the permissions or ownership of a file/directory
* send mails or notifications

Ansible provides a [large amount](http://docs.ansible.com/ansible/latest/list_of_all_modules.html) of built-in modules, and even allows to write own modules. There are already modules for most of the standard Unix commands.

**This is how a simplified version of our deployment distribution playbook looks like:**

- hosts: production

vars\_files:

- vars-common.yaml

strategy: free

tasks:

- name: copy bundle to remote host

copy: src={{ filepath\_master\_bundle }} dest={{ filepath\_remote\_bundle }}

- name: create remote bundle directory

file: path={{ path\_remote\_bundles }} state=directory mode=0775

- name: extract bundle to target directory

unarchive: copy=no src={{ filepath\_remote\_bundle }} dest={{ path\_remote\_bundles }}

The first line just defines which **host list** is used for this playbook run. In this case, we want to distribute our bundle to all production hosts. The list itself is defined in a global *inventory* file on the master.

With the next lines, some **variable** definitions from a separate file (vars-common.yaml) are imported. Variables are a great method to hold definitions like file names etc. at a central place, so that they can be used from multiple playbooks.

The **strategy** defines the order and parallelity Ansible uses when executing the tasks. While tasks are executed in strict order on each individual host, this is not necessarily true for the hosts. With the *free* strategy we are using here, Ansible can decide on its own how many hosts are run parallely, and when tasks are actually executed. It might even happen that the first couple of hosts have already finished, while some hosts have not even yet started.

At the end, the **tasks** are defined. First we *copy* the bundle with the version from the master to the slave. With the *file* module, we ensure that the target directory (where we intend to extract the bundle to) exists and has the proper permissions. Finally, we *unarchive* the bundle to the target directory (tar extract).

When the Ansible playbook run on the master has finished, all slaves have a new directory with the contents of the bundle we want to deploy. The slaves are now ready to switch to the new version.

**Activating the New Version**

To activate the new version, another playbook is executed. This playbook follows a different strategy. On one hand, we want the change to roll out nearly simultanously. On the other hand, if something goes wrong during activation, or the new version does not run properly, we want to have enough production machines running the old version in case we need to abort the activation.

Again, here is a simplified version of the playbook for the activation:

- hosts: production

vars\_files:

- vars-common.yaml

strategy: linear

serial: "33%"

max\_fail\_percentage: 33

tasks:

- name: run initial smoke test to make sure everything is OK

script: "{{ web\_smoke\_test\_command }} {{ inventory\_hostname }}"

- name: web server config test

shell: "{{ web\_server\_test\_config\_command }}"

- name: link production directory to new bundle directory

file: src={{ path\_remote\_bundle }} dest={{ path\_remote\_prod }} state=link }}"

- name: web server graceful reload

shell: "{{ web\_server\_graceful\_command }} warn=no"

- name: run smoke test to make sure the new version is running

script: "{{ web\_smoke\_test\_command }} {{ inventory\_hostname }}"

The *linear* strategy will execute the tasks on batches of hosts parallely. In this case, we chose a batch size of 33%. The next batch will start only after the previous batch has finished. Additionally, we specify a *maximum fail percentage* of 33%. That means, if at least 33% of the hosts failed to run the tasks without error, the execution will stop and the remaining hosts are skipped. So if the first batch fails, further activation of the new version is stopped, and the remaining 66% can still deliver the previous version.

The actual activation tasks first do a smoke test to ensure the application is in a proper state before we begin. Then we verify the web server configuration to avoid that we restart the server with an invalid configuration. Then we point the production application directory to the bundle path where we extracted the files in the distribution playbook run by setting a softlink with the *file* module. After the reload of the web server, a final smoke test ensures the application is still running properly with the new version.

**Rollback**

In case we need to roll back to a previous version, we just need to run the activation playbook and pass the bundle directory of that version in the *path\_remote\_bundle* variable. As we keep older bundles on the production servers for some time, the distribution playbook (which takes much more time than the activation playbook) is obsolete.

**Conclusion**

Ansible is a great tool for automating jobs like software deployment. There are hundreds of module available, and many more features like conditionals, loops etc. that were not covered in this article.