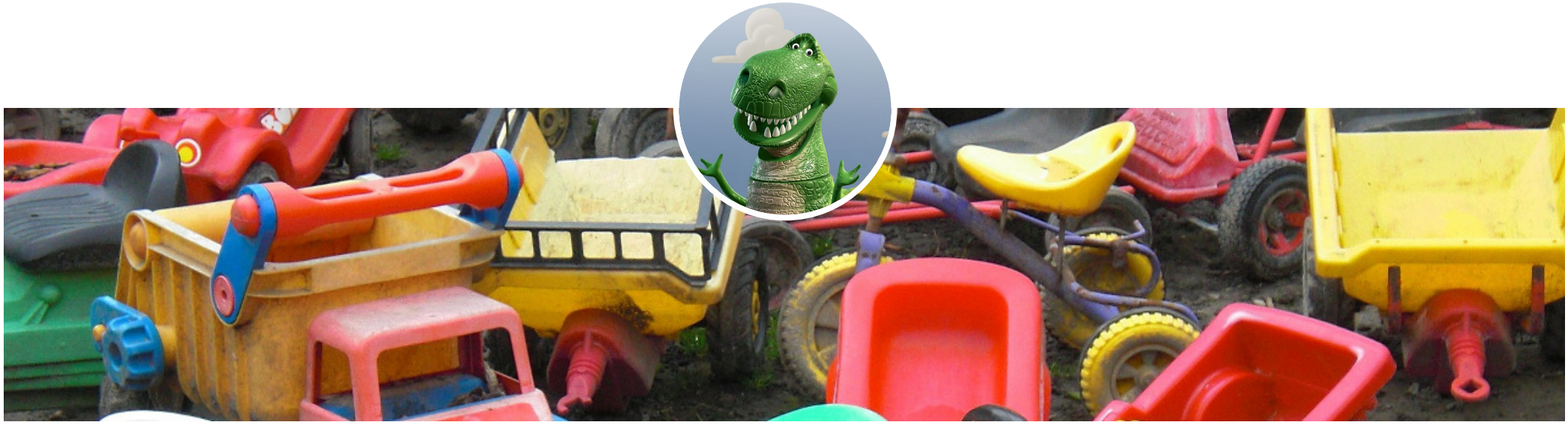

Image credit: Paramjeet

ANSIBLE

Laying out roles, inventories and playbooks

BY MICHEL BLANC JULY 02, 2015 TWEET LIKE +1

I have been writing playbooks for quite a while now. Along the way, I went through various stages, and used different ways to layout Ansible files. I guess that after going down this trial and error path, I finally came up with something I will stick to.

I am not saying that this is the be-all and end-all of Ansible files layout but may be it will fast forward you to a saner file layout, and you'll be able to move on from there. This post will probably help you if you are new to Ansible, trying to figure out what to put and where. I hope it will prove usefull if you have some Ansible experience too.

Some terminology

In this post, I will mostly talk about 3 things: roles, inventories and playbooks. Other items do exist (plays, tasks, …) but those 3 elements shape the big picture of the layout.

Roles

A role is a collection of tasks and templates (among other things, but those are the most common) focused on one very specific goal. For instance, you can have a role that installs nginx, another that deploys ssh keys for admins, etc…

Nginx role will install and configure nginx. Nothing else. It won't create DNS entries, trim logs, add a ftp server or anything. It just installs nginx. Period.

Inventories

An inventory is a list of hosts, eventually assembled into groups, on which you will run ansible playbooks. Ansible automatically puts all defined hosts in the aptly named group `all`.

For instance, you could have hosts `www1` and `www2`, assembled in group `webservers`, and later reference the group or individual hosts, depending on your needs.

Inventories can also come with variables applied to hosts or groups (including `all`).

Inventories can be dynamic. If the inventory file is executable, Ansible will run it and use its output as the inventory (note that, in this case, the format is not the same as static inventory).

You can of course have multiple inventories, segregated from each other. We will take advantage from this later on.

Playbooks

The last piece of the puzzle is the playbook. The playbook is the pivot between and inventory and roles. This is where you basically tell Ansible: *please install roles foo, bar and baz on machines alice, bob and charlie*.

Role layout

Role layout is pretty well documented at Ansible website. A role contains several directories. All directories are optional besides `tasks`. For each directory, the entry point is `main.yml`. Thus, the only compulsory file in a role is `tasks/main.yml`.

```
ansible-foobar/
├── defaults
│   └── main.yml
├── files
├── handlers
│   └── main.yml
├── meta
│   └── main.yml
├── tasks
│   ├── check_vars.yml
│   ├── foobar.yml
│   └── main.yml
└── templates
    └── foobar.conf.j2
```

Let's cover briefly the layout an see the function of each file and directory.

`defaults/main.yml`

This directory contains defaults for variables used in roles. I encourage you to define every variable used in your role, for several reasons:

- this file will be a nice and always up to date reference list of settings configuration in your roles
- having configured variables will prevent your role failing in an uncontrolled way (more on this later).

If some of these variables are used in templates to generate config files, I highly encourage you to use your target OS defaults. The principle of least surprise should apply here.

Best practices assumes that you are using *pseudo-namespacing* for your role's variables (e.g. for role `foobar`, all variables should begin with `foobar_`) to avoid collisions with other roles.

`files/`

This directory holds files that do not require Jinja interpolation, and can be copied as-is on the remote nodes.

`handlers/main.yml`

This is where you define handlers that get notified by tasks. Handlers are just standard tasks. You can use `include` in this file if you want to separate handlers (for different OSes versions for instances), but try to keep the file number as low as possible so you don't end up hunting down stuff everywhere.

If your handler restarts any service, you have to make sure that the service config file is valid before attempting to restart it. Some daemons allow this (e.g. nginx, haproxy, apache). If your service does not, provide some fallback mechanism. You don't want your playbook to screw up your running system because you typoed a configuration variable. See the `validate` option in the template module.

Note that handlers are just standard tasks.

`meta/main.yml`

This metadata has (AFAIK) only two variables:

- `galaxy_info` : meta information for galaxy about your role. You just don't need this if you don intend to push your role to Galaxy. For details on the format, see TODO: find ref
- `dependencies` : what roles this role depends on.

The latter is of utmost importance, and setting it right deserves a blog post on it's own. Until then, the rule of thumb to remember is to **only include compulsory role dependencies for the target host**.

This means that adding `nginx` dependency in a `php-fpm` role sounds perfectly reasonable[1]. However, adding a `mysql` dependency to your web application role is not, because `mysql` can be deployed on another server.

`tasks/main.yml`

This file is the tasks entry point. However, it should be mostly empty. Why ? Because you want to use Ansible tags. Tags are a great way to limit task execution for an Ansible run, where only tagged tasks are run.

For instance, in a playbook that deploys your application, you could choose to run only tasks regarding nginx.

The problem is that tagging every task in `main.yml` would be cumbersome, error prone, and clutter the code unnecessarily.

The best way to tag all your tasks is to include your real task file from `tasks/main.yml` and tag the whole file:

```
- include: foobar.yml tags=foobar
```

Here, I name the real task file `foobar.yml` with the same name as the role (quite handy with `find` or `locate`; no need to guess which `main.yml` you are looking for) and apply the tag `foobar` to all tasks in the role.

You can repeat this if you have a big list of tasks and want to split them in several files. You could, for instance, separate configuration and installation matters, and add another specific tag for each of them:

```
- include: foobar-install.yml tags=foobar,foobar:install
- include: foobar-config.yml tags=foobar,foobar:config
```

Here I added two tags to the installation part (`foobar` and `foobar:install`), and two for the configuration part (`foobar` and `foobar:config`).

Note that the `:` between, for instance, `foobar` and `config` has no meaning. Ansible treats tags as dumb strings. It is just a personnal convention (Redis like) for refining tags.

With this setup, you could run only the configuration part of your role by issuing:

```
ansible-playbook playbook.yml -t foobar:config
```

The `-t` and `-l` combination is a very powerful weapon to target a specific host with a precise change (think of this as pointing to a matrix cell targetting host (i.e. row) and tag (i.e. column)).

`tasks/check_vars.yml`

I use this file to ensure that required variables are defined.

```
#
# Checking that required variables are set
#
- name: Checking that required variables are set
  fail: msg="{{ item }} is not defined"
  when: not {{ item }}
  with_items:
    - foobar_database
    - foobar_deploy_user
```

Then, include this file in `tasks/main.yml`:

```
- include: check_vars.yml tags=foobar,foobar:check,check
- include: foobar.yml tags=foobar
```

`templates/*`

This is the place where templates (i.e. files with interpolated variables goes). While this is not necessary, I often reference them using a relative path like so:

```
- name: Template foo
  template:
    src: "../templates/foo.conf.j2"
    dest: /some/place/in/the/node/filesystem/foo.conf
```

The goal of using relative path is to be able to hit `gf` in Vim and open the file directly. You can get rid of that and just use `src: foo.conf.j2`. It is just a readability/convenience tradeoff.

The file name I use is the **intended filename at the destination**, appended with `.j2` so it is clear that it is a Jinja2 template, and easier to search (`find` or `locate`).

Some folks like to replicate the destination hierarchy (e.g. `src: etc/sysconfig /network-scripts/ifcfg-ethx.cfg.j2`). This is a matter of taste, but personaly I don't see the point of having those deep hierarchies in the role if the naming is correct.

`vars/main.yml`

It is sometimes difficult to grok the difference between `vars/main.yml` and `defaults/main.yml`. After all, they both contain variable assignements.

I do not always use a `vars/main.yml`, but when I do, I put “constants like” variables in it. These are variables that are not intended to be overriden.

For instance the github repository for a particular piece of code (e.g. your web application) will certainly go there. However, the version you want to deploy won’t.

All in all, it is just a mechanism to take those values out of tasks files readability and role life cycle.

Inventories and playbook layout

A playbook glues together roles and inventories. Thus playbooks depend on roles and inventories. But while you have mechanisms to list roles requirements in a playbook, you don’t have any for inventories.

Since the playbook can not live without the targeted inventories I include my inventories in my playbooks.

```
playbook-foobar/
├── ansible.cfg
├── requirements.yml
├── .imported_roles/
├── inventories
|   ├── development
|   |   ├── group_vars
```

```
│   │   │   └── all
│   │   └── hosts
│   ├── integration
│   │   ├── group_vars
│   │   │   └── all
│   │   └── hosts
│   └── production
│       ├── group_vars
│       │   └── all
│       └── hosts
├── site.yml
└── playbooks
    ├── database.yml
    └── stuff.yml
```

`ansible.cfg`, `.imported_roles/` and `requirements.yml`

This file controles ansible behaviour. You can have one in `/etc/ansible` or as a personal dotfile (`~/.ansible.cfg`). Adding an `ansible.cfg` file in the playbook root will ensure that the required settings for the playbook to run are really there. The precedence order for Ansible config files is[2]:

1. ANSIBLE_CONFIG (an environment variable pointing to a file)
2. ansible.cfg (in the current directory)
3. .ansible.cfg (in the home directory)
4. /etc/ansible/ansible.cfg

Ansible will use the first config file found.

In this config file, I always set at least two options:

```
hostfile = ./inventories/dev
roles_path = ./.imported_roles:/some/dev/place/with/roles
```

The first one (`hostfile`) sets which inventory Ansible will use. More explanations will come below.

The second one set the path where Ansible will look for roles. I typically set two directories here (separated by `:` , like shell's `PATH`):

- the first directory will be used by Ansible galaxy hold imported files. I set it to `.imported_roles` but the name doesn't matter. Don't forget to add it to your playbook's `.gitignore` though.

- the second directory points to my roles developmenent directory path

The advantages for this setup are two fold: first, you have a dedicated path, ignored by your SCM, where you will download roles. The roles will be searched there first. Secondly, if a role is not found, it will be searched in your role development directory. This let you hack on your roles while writing a playbook. You don't need to go through a *commit/push/install* cycle when you are coding your roles for this playbook.

Roles dependencies for your playbook are listed in `requirements.yml` and can be installed with `ansible-galaxy install -r requirements.yml`:

```
# Role on galaxy
- you.rolename
# Public role on github
- name: role-public
  src: https://github.com/erasme/role-public.git
# Private role on github
- name: role-private
  src: git+ssh://git@github.com/you/role-private.git
```

`inventories/`

This directory holds all inventories you want to apply your playbook too. The most common pattern is to use per-environment inventories: one for `development`, one for `integration`, another for `production`, etc…

Of course, the `hostfile` variable in `ansible.cfg` should point to `development` to avoid accidentaly messing with production. Executing the playbook on non-development inventories will force you tu use the `-i`, which is a good safety measure.

While you can define variables in groups (in `group_vars`) and hosts (`host_vars`), you should stuff as much variables as possible in `group_vars/all`. The rationale is that it is much easier to find a variable when a single file is involved.

Variables scattered in a dozen of files are *not* manageable.

And when you'll want to create an additional inventory (e.g. create `production` from `development`), it will be much easier to change a single file and set the variables to proper values than to do the same in several files.

Note that `group_vars/all` can be a directory containing several files. I usually split variables in a clear text file (`group_vars/all/all`) and a ciphered one (`group_vars/all_secret`) using the transparent vaulting techniques described in
this post.

`site.yml` and `playbooks/`

This directory contains the playbookks themselves. I always create a "master" playbook called `site.yml` in the playbook root directory, which includes all other playbooks located in `playbooks/`. For instance:

```
#!/usr/bin/env ansible-playbook

- include: playbooks/database.yml
- include: playbooks/stuff.yml
```

The rationale is to be able to use `ansible-pull` easily if needed (`ansible- pull`, by default, tries to execute a playbook called `site.yml`). The other point is to split playbook in related parts.

For instance, you could have a playbook the takes care of setting up the database, another that will set the OS level stuff (e.g. ssh keys, firewalling, …), another one that takes care of deploying your web application, etc… When needed, You can use all the playbooks at once with `site.yml`, or just focus on a specific problem running the appropriate playbook (no need to run the ssh-key setup if you're just deploying the latest version of your web application).

The shebang line at the top of the file (`#!/usr/bin/env ansible-playbook`) will make the playbook directly executable (adjust `ansible-playbook` path and `chmod +x` the playbook file).

Layout Antipatterns

When I started using Ansible, I cumulated several antipatterns at the same time: trying to emcompass all my infrastructure in a single inventory containing per-host fine grained variables, used in a single playbook, without using any role.

While this sounds feasible, it is doomed to failure unless you manage a very small infrastructure. Let's zoom in briefly on each mistake.

Trying to encompass all your infrastructure in one playbook

Is is tempting to aim for a one-liner that will magically deploy all your infrastructure in one shot. This gives you some bragging rights at your next meetup, and feels like the ultimate sysadmin masterpiece.

However, it has many drawbacks:

- it will be slow: do you really want to run a playbook over dozens of more tasks or roles, just to change an entry in `/etc/hosts` ? Yes, there are workaround for this, but it will require some command line magic, a lot of thinking.

- it mixes bananas and apples: you should strive for separation of concerns in your playbooks if you want be able to read them (and, as a consequence, maintain them).

As a consequence, your infrastructure code will be unnecessary hard to test and maintain.

Per-host fine grained variables

This is a corolary of the previous antipattern: when you try to encompass your whole infrastructure, you start to think, inheritance, variables overriding and refining.

And while doing this, you add considerable complexity to your inventories. It is very hard to track down variables definitions when you overrides them in `group_vars/some_group` , `group_vars/all` , `hosts_vars/machine` , role defaults, …

Now this can get even worse when you use the `hash_behavior: merge` Ansible configuration setting: it introduces more confusion, and makes your Ansible work potentially unshareable with people using `hash_behaviour: replace` . Since I am guilty on this one, it is time to make some apologies. Sorry folks. Michael DeHaan did not like it, and he was right.

Single playbook

A single playbook relates to the first Sin again, but also applies to more focused playbooks where you only deploy one thing. Splitting your playbooks between various logically related roles will fasten your deployments. Again, why running ssh key distribution, storage cluster deployment, web stack, middlewares and application when you just change the color of a button in your web app ?

Split your playbook in related parts that reflects your stack architecture. They will be faster and easier to use.

No roles (tasks only)

Well, this is obvious. Even if you don't want to share, make roles and strive for code reuse. Reused code will save you time of course, but it is also battle tested since it is used more frequently.

Tasks-only playbook can be used for a quick hit and run, solving a transient problem that doesn't offer any code reuse opportunities.

I also try to avoid tasks along roles in playbooks: this hurts the abstraction level you manage to build using roles. When thinking in terms of roles, you don't need to think about the nitty gritty details of the roles when reading your playbooks. If your roles are thouroughly tested, you can read your infrastructure in seconds. Add tasks to the mix, and you loose this superpower.

1. Yes, you could separate your application server (e.g. php-fpm) and put it on a different machine than your webserver, it ll depends on your local context. ↩

2. http://docs.ansible.com/intro_configuration.html ↩

Previous