Chapter 6 - Playbook Organization - Roles, Includes, and Imports

So far, we've used fairly straightforward examples in this book. Most examples are created for a particular server, and are in one long playbook.

Ansible is flexible when it comes to organizing tasks in more efficient ways so you can make playbooks more maintainable, reusable, and powerful. We'll look at two ways to split up tasks more efficiently using includes and roles, and we'll explore Ansible Galaxy, a repository of some community-maintained roles that help configure common packages and applications.

Imports

We've already seen one of the most basic ways of including other files in Chapter 4, when `vars_files` was used to place variables into a separate `vars.yml` file instead of inline with the playbook:

```
- hosts: all

  vars_files:
    - vars.yml
```

Tasks can easily be included in a similar way. In the `tasks:` section of your playbook, you can add `import_tasks` directives like so:

```
tasks:
  - import_tasks: imported-tasks.yml
```

Just like with variable include files, tasks are formatted in a flat list in the included file. As an example, the `imported-tasks.yml` could look like this:

```
---
- name: Add profile info for user.
  copy:
    src: example_profile
    dest: "/home/{{ username }}/.profile"
    owner: "{{ username }}"
    group: "{{ username }}"
    mode: 0744

- name: Add private keys for user.
  copy:
    src: "{{ item.src }}"
    dest: "/home/{{ username }}/.ssh/{{ item.dest }}"
    owner: "{{ username }}"
    group: "{{ username }}"
    mode: 0600
  with_items: "{{ ssh_private_keys }}"

- name: Restart example service.
  service: name=example state=restarted
```

In this case, you'd probably want to name the file `user.yml`, since it's used to configure a user account and restart some service. Now, in this and any other playbook that provisions or configures a server, if you want to configure a particular user's account, add the following in your playbook's `tasks` section:

```
tasks:
  - import_tasks: user.yml
```

We used `{{ username }}` and `{{ ssh_private_keys }}` variables in this include file instead of hard-coded values so we could make this include file reusable. You could define the variables in your playbook's inline variables or an included variables file, but Ansible also lets you pass variables directly into includes using normal YAML syntax. For example:

```
tasks:
  - import_tasks: user.yml
    vars:
      username: johndoe
      ssh_private_keys:
        - { src: /path/to/johndoe/key1, dest: id_rsa }
        - { src: /path/to/johndoe/key2, dest: id_rsa_2 }
  - import_tasks: user.yml
    vars:
      username: janedoe
      ssh_private_keys:
        - { src: /path/to/janedoe/key1, dest: id_rsa }
        - { src: /path/to/janedoe/key2, dest: id_rsa_2 }
```

Imported files can even import other files, so you could have something like the following:

```
tasks:
  - import_tasks: user-config.yml
```

inside `user-config.yml`

```
- import_tasks: ssh-setup.yml
```

Includes

If you use `import_tasks`, Ansible statically imports the task file as if it were part of the main playbook, once, before the Ansible play is executed.

If you need to have included tasks that are *dynamic*—that is, they need to do different things depending on how the rest of the playbook runs—then you can use `include_-tasks` rather than `import_tasks`.

As an example, in one of my Ansible projects, I have a task file `log_paths.yml` with the following:

```
---
- name: Check for existing log files in dynamic log_file_paths variable.
  find:
    paths: "{{ item }}"
    patterns: '*.log'
  register: found_log_file_paths
  with_items: "{{ log_file_paths }}"
```

In this case, the `log_file_paths` variable is set by a task earlier in my playbook—so this include file wouldn't be able to know the value of that variable until the playbook has already partly completed.

So when I include this task file, I have to do so dynamically, for example:

```
- include_tasks: log_paths.yml
```

> Early on, Ansible only had static `include` available for task inclusion, but as playbooks became more complex, people need to be able to include tasks that were processed when run (instead of added to the list of tasks before the play started running). So Ansible 2.1 introduced the `static` flag for `include:`. This worked, but overloaded the use of one keyword, so in Ansible 2.4, the use of `include:` was deprecated and you should use `import_tasks` if your tasks can basically be inlined before the playbook runs, or `include_tasks` if the tasks might need to be more dynamic (e.g. registering and reacting to a new registered variable).

Dynamic includes

Until Ansible 2.0, `includes` were always processed when your playbook run started (just like `import_tasks` behaves now), so you couldn't do things like load a particular include when some condition was met. Ansible 2.0 and later evaluates includes during playbook execution, so you can do something like the following:

```
# Include extra tasks file, only if it's present at runtime.
- name: Check if extra_tasks.yml is present.
  stat: path=tasks/extra-tasks.yml
  register: extra_tasks_file
  connection: local

- include_tasks: tasks/extra-tasks.yml
  when: extra_tasks_file.stat.exists
```

If the file `tasks/extra-tasks.yml` is not present, Ansible skips the `include_tasks`. You can even use a `with_items` loop (or any other `with_*` loop) with includes. Includes evaluated during playback execution can make your playbooks much more flexible!

Handler imports and includes

Handlers can be imported or included just like tasks, within a playbook's `handlers` section. For example:

```
handlers:
  - import_tasks: handlers.yml
```

This can be helpful in limiting the noise in your main playbook, since handlers are usually used for things like restarting services or loading a configuration, and can distract from the playbook's primary purpose.

Playbook imports

Playbooks can even be included in other playbooks, using the same `import` syntax in the top level of your playbook (though for playbooks, you only have `import_-playbook` available, as they cannot be dynamic like task includes). For example, if you have two playbooks—one to set up your webservers (`web.yml`), and one to set up your database servers (`db.yml`), you could use the following playbook to run both at the same time:

```
- hosts: all
  remote_user: root

  tasks:
    [...]

- import_playbook: web.yml
- import_playbook: db.yml
```

This way, you can create playbooks to configure all the servers in your infrastructure, then create a master playbook that includes each of the individual playbooks. When you want to initialize your infrastructure, make changes across your entire fleet of servers, or check to make sure their configuration matches your playbook definitions, you can run one `ansible-playbook` command!

Complete includes example

What if I told you we could remake the 137-line Drupal LAMP server playbook from Chapter 4 in just 21 lines? With includes, it's easy; just break out each of the sets of tasks into their own include files, and you'll end up with a main playbook like this:

```
---
- hosts: all

  vars_files:
    - vars.yml

  pre_tasks:
    - name: Update apt cache if needed.
      apt: update_cache=yes cache_valid_time=3600

  handlers:
    - import_tasks: handlers/handlers.yml

  tasks:
    - import_tasks: tasks/common.yml
    - import_tasks: tasks/apache.yml
    - import_tasks: tasks/php.yml
    - import_tasks: tasks/mysql.yml
    - import_tasks: tasks/composer.yml
    - import_tasks: tasks/drush.yml
    - import_tasks: tasks/drupal.yml
```

All you need to do is create two new folders in the same folder where you saved the Drupal `playbook.yml` file, `handlers` and `tasks`, then create files inside for each section of the playbook.

For example, inside `handlers/handlers.yml`, you'd have:

```
---
- name: restart apache
  service: name=apache2 state=restarted
```

And inside `tasks/drush.yml`:

```
---
- name: Check out drush 8.x branch.
  git:
    repo: https://github.com/drush-ops/drush.git
    version: 8.x
    dest: /opt/drush

- name: Install Drush dependencies with Composer."
  shell: >
    /usr/local/bin/composer install
    chdir=/opt/drush
    creates=/opt/drush/vendor/autoload.php

- name: Create drush bin symlink.
  file:
    src: /opt/drush/drush
```

```
17       dest: /usr/local/bin/drush
18       state: link
```

Separating all the tasks into separate task files means you'll have more files to manage for your playbook, but it helps keep the main playbook more compact. It's easier to see all the installation and configuration steps the playbook contains, and it separates tasks into individual, maintainable groupings. Instead of having to browse one playbook with twenty-three separate tasks, you now maintain eight included files with two to five tasks, each.

It's much easier to maintain small groupings of related tasks than one long playbook. However, there's no reason to try to *start* writing a playbook with lots of individual includes. Most of the time, it's best to start with a monolithic playbook while you're working on the setup and configuration details, then move sets of tasks out to included files after you start seeing logical groupings.

You can also use tags (demonstrated in the previous chapter) to limit the playbook run to a certain task file. Using the above example, if you wanted to add a 'drush' tag to the included drush file (so you could run `ansible-playbook playbook.yml --tags=drush` and only run the drush tasks), you can change line 20 to the following:

```
20   - import_tasks: tasks/drush.yml tags=drush
```

You can find the entire example Drupal LAMP server playbook using include files in this book's code repository at https://github.com/geerlingguy/ansible-for-devops[68], in the `includes` directory.

You can't use variables for task include file names when using `import_tasks` (like you could with `include_vars` directives, e.g. `include_vars: "{{ ansible_os_family }}.yml"` as a task, or with `vars_files`), but you can when using `include_tasks` (dynamically). In either case, it might be easier to accomplish conditional task inclusion using a different playbook structure, or roles, which we will discuss next.

[68]https://github.com/geerlingguy/ansible-for-devops

Roles

Including playbooks inside other playbooks makes your playbook organization a little more sane, but once you start wrapping up your entire infrastructure's configuration in playbooks, you might end up with something resembling Russian nesting dolls.

Wouldn't it be nice if there were a way to take bits of related configuration, and package them together nicely? Additionally, what if we could take these packages (often configuring the same thing on many different servers) and make them flexible so that we can use the same package throughout our infrastructure, with slightly different settings on individual servers or groups of servers?

Ansible Roles can do all that and more!

Let's dive into what makes an Ansible role by taking one of the playbook examples from Chapter 4 and splitting it into a more flexible structure using roles.

Role scaffolding

Instead of requiring you to explicitly include certain files and playbooks in a role, Ansible automatically includes any `main.yml` files inside specific directories that make up the role.

There are only two directories required to make a working Ansible role:

```
role_name/
  meta/
  tasks/
```

If you create a directory structure like the one shown above, with a `main.yml` file in each directory, Ansible will run all the tasks defined in `tasks/main.yml` if you call the role from your playbook using the following syntax:

```
---
- hosts: all
  roles:
    - role_name
```

Your roles can live in a couple different places: the default global Ansible role path (configurable in `/etc/ansible/ansible.cfg`), or a `roles` folder in the same directory as your main playbook file.

Another simple way to build the scaffolding for a role is to use the command: `ansible-galaxy init role_name`. Running this command creates an example role in the current working directory, which you can modify to suit your needs. Using the `init` command also ensures the role is structured correctly in case you want to someday contribute the role to Ansible Galaxy.

Building your first role

Let's clean up the Node.js server example from Chapter four, and break out one of the main parts of the configuration—installing Node.js and any required npm modules.

Create a `roles` folder in the same directory as the main `playbook.yml` file like we created in Chapter 4's first example. Inside the `roles` folder, create a new folder: `nodejs` (which will be our role's name). Create two folders inside the `nodejs` role directory: `meta` and `tasks`.

Inside the `meta` folder, add a simple `main.yml` file with the following contents:

```
---
dependencies: []
```

The meta information for your role is defined in this file. In basic examples and simple roles, you just need to list any role dependencies (other roles that are required to be run before the current role can do its work). You can add more to this file to describe your role to Ansible and to Ansible Galaxy, but we'll dive deeper into meta information later. For now, save the file and head over to the `tasks` folder.

Create a `main.yml` file in this folder, and add the following contents (basically copying and pasting the configuration from the Chapter 4 example):

```
---
- name: Install Node.js (npm plus all its dependencies).
  yum: name=npm state=present enablerepo=epel

- name: Install forever module (to run our Node.js app).
  npm: name=forever global=yes state=present
```

The Node.js directory structure should now look like the following:

```
nodejs-app/
  app/
    app.js
    package.json
  playbook.yml
  roles/
    nodejs/
      meta/
        main.yml
      tasks/
        main.yml
```

You now have a complete Ansible role that you can use in your node.js server configuration playbook. Delete the Node.js app installation lines from `playbook.yml`, and reformat the playbook so the other tasks run first (in a `pre_tasks:` section instead of `tasks:`), then the role, then the rest of the tasks (in the main `tasks:` section). Something like:

```
pre_tasks:
  # EPEL/GPG setup, firewall configuration...

roles:
  - nodejs

tasks:
  # Node.js app deployment tasks...
```

You can view the full example of this playbook in the ansible-for-devops code repository[69].

Once you finish reformatting the main playbook, everything will run exactly the same during an `ansible-playbook` run, with the exception of the tasks inside the `nodejs` role being prefixed with `nodejs | [Task name here]`.

This little bit of extra data shown during playbook runs is useful because it automatically prefixes tasks with the role that provides them, without you having to add in descriptions as part of the `name` values of the tasks.

Our role isn't all that helpful at this point, though, because it still does only one thing, and it's not really flexible enough to be used on other servers that might need different Node.js modules to be installed.

More flexibility with role vars and defaults

To make our role more flexible, we can make it use a list of npm modules instead of a hardcoded value, then allow playbooks using the role to provide their own module list variable to override our role's default list.

When running a role's tasks, Ansible picks up variables defined in a role's `vars/main.yml` file and `defaults/main.yml` (I'll get to the differences between the two later), but will allow your playbooks to override the defaults or other role-provided variables if you want.

Modify the `tasks/main.yml` file to use a list variable and iterate through the list to install as many packages as your playbook wants:

[69]https://github.com/geerlingguy/ansible-for-devops/blob/master/nodejs-role/

```
---
- name: Install Node.js (npm plus all its dependencies).
  yum: name=npm state=present enablerepo=epel

- name: Install npm modules required by our app.
  npm: name={{ item }} global=yes state=present
  with_items: "{{ node_npm_modules }}"
```

Let's provide a sane default for the new `node_npm_modules` variable in `defaults/main.yml`:

```
---
node_npm_modules:
  - forever
```

Now, if you run the playbook as-is, it will still do the exact same thing—install the `forever` module. But since the role is more flexible, we could create a new playbook like our first, but add a variable (either in a `vars` section or in an included file via `vars_files`) to override the default, like so:

```
node_npm_modules:
  - forever
  - async
  - request
```

When you run the playbook with this custom variable (we didn't change *anything* with our `nodejs` role), all three of the above npm modules will be installed.

Hopefully you're beginning to see how this can be powerful!

Imagine if you had a playbook structure like:

```
---
- hosts: appservers
  roles:
    - yum-repo-setup
    - firewall
    - nodejs
    - app-deploy
```

Each one of the roles lives in its own isolated world, and can be shared with other servers and groups of servers in your infrastructure.

- A `yum-repo-setup` role could enable certain repositories and import their GPG keys.
- A `firewall` role could have per-server or per-inventory-group options for ports and services to allow or deny.
- An `app-deploy` role could deploy your app to a directory (configurable per-server) and set certain app options per-server or per-group.

These things are easy to manage when you have small bits of functionality separated into different roles. Instead of managing 100+ lines of playbook tasks, and manually prefixing every `name:` with something like "Common |" or "App Deploy |", you now manage a few roles with 10-20 lines of YAML each.

On top of that, when you're building your main playbooks, they can be extremely simple (like the above example), enabling you to see *everything* being configured and deployed on a particular server without scrolling through dozens of included playbook files and hundreds of tasks.

Variable precedence: Note that Ansible handles variables placed in included files in `defaults` with less precedence than those placed in `vars`. If you have certain variables you need to allow hosts/playbooks to easily override, you should probably put them into `defaults`. If they are common variables that should almost always be the values defined in your role, put them into `vars`. For more on variable precedence, see the aptly-named "Variable Precedence" section in the previous chapter.

Other role parts: handlers, files, and templates

Handlers

In one of the prior examples, we introduced handlers—tasks that could be called via the `notify` option after any playbook task resulted in a change—and an example handler for restarting Apache was given:

```
handlers:
  - name: restart apache
    service: name=apache2 state=restarted
```

In Ansible roles, handlers are first-class citizens, alongside tasks, variables, and other configuration. You can store handlers directly inside a `main.yml` file inside a role's `handlers` directory. So if we had a role for Apache configuration, our `handlers/main.yml` file could look like this:

```
---
- name: restart apache
  service: name=apache2 state=restarted
```

You can call handlers defined in a role's handlers folder just like those included directly in your playbooks (e.g. `notify: restart apache`).

Files and Templates

For the following examples, let's assume our role is structured with files and templates inside `files` and `templates` directories, respectively:

```
roles/
  example/
    files/
      example.conf
    meta/
      main.yml
    templates/
      example.xml.j2
    tasks/
      main.yml
```

when copying a file directly to the server, add the filename or the full path from within a role's `files` directory, like so:

```
- name: Copy configuration file to server directly.
  copy:
    src: example.conf
    dest: /etc/myapp/example.conf
    mode: 0644
```

Similarly, when specifying a template, add the filename or the full path from within a role's `templates` directory, like so:

```
- name: Copy configuration file to server using a template.
  template:
    src: example.xml.j2
    dest: /etc/myapp/example.xml
    mode: 0644
```

The `copy` module copies files from within the module's `files` folder, and the `template` module runs given template files through the Jinja templating engine, merging in any variables available during your playbook run before copying the file to the server.

Organizing more complex and cross-platform roles

For simple package installation and configuration roles, you can get by with placing all tasks, variables, and handlers directly in the respective `main.yml` file Ansible automatically loads. But you can also *include* other files from within a role's `main.yml` files if needed.

As a rule of thumb, I keep my playbook and role task files under 100 lines of YAML if at all possible. It's easier to keep the entire set of tasks in my head while making changes or fixing bugs. If I start nearing that limit, I usually split the tasks into logical groupings, and include files from the `main.yml` file.

Let's take a look at the way my `geerlingguy.apache` role is set up (it's available on Ansible Galaxy[70] and can be downloaded to your roles directory with the command `ansible-galaxy install geerlingguy.apache`; we'll discuss Ansible Galaxy itself later).

Initially, the role's main `tasks/main.yml` file looked something like the following (generally speaking):

```
- name: Ensure Apache is installed (via apt).

- name: Configure Apache with lineinfile.

- name: Enable Apache modules.
```

Soon after creating the role, though, I wanted to make the role work with both Debian and RHEL hosts. I could've added two sets of tasks in the `main.yml` file, resulting in twice the number of tasks and a bunch of extra `when` statements:

```
- name: Ensure Apache is installed (via apt).
  when: ansible_os_family == 'Debian'

- name: Ensure Apache is installed (via yum).
  when: ansible_os_family == 'RedHat'

- name: Configure Apache with lineinfile (Debian).
  when: ansible_os_family == 'Debian'

- name: Configure Apache with lineinfile (RHEL).
```

[70] https://galaxy.ansible.com/geerlingguy/apache/

```
  when: ansible_os_family == 'RedHat'

- name: Enable Apache modules (Debian).
  when: ansible_os_family == 'Debian'

- name: Other OS-agnostic tasks...
```

If I had gone this route, and continued with the rest of the playbook tasks in one file, I would've quickly surpassed my informal 100-line limit. So I chose to use includes in my main tasks file:

```
- name: Include OS-specific variables.
  include_vars: "{{ ansible_os_family }}.yml"

- name: Include OS-specific setup tasks.
  include_tasks: setup-{{ ansible_os_family }}.yml

- name: Other OS-agnostic tasks...
```

Two important things to notice about this style of distribution-specific inclusion:

1. When including vars and tasks files (with `include_vars` or `include_tasks`), you can actually *use variables in the name of the file*. This is handy in many situations; here we're including a vars file in the format `distribution_name.yml`. For our purposes, since the role will be used on Debian and RHEL-based hosts, we can create `Debian.yml` and `RedHat.yml` files in our role's `defaults` and `vars` folders, and put distribution-specific variables there.
2. For the tasks, we include tasks files in the role's `tasks` directory, for example `setup-Debian.yml` or `setup-RedHat.yml`.

After setting things up this way, I put RHEL and CentOS-specific tasks (like `yum` tasks) into `tasks/setup-RedHat.yml`, and Debian and Ubuntu-specific tasks (like `apt` tasks) into `tasks/setup-Debian.yml`. There are other ways of making roles work cross-platform, but using distribution-specific variables files and included task files is one of the simplest.

Now this Apache role can be used across different distributions, and with clever usage of variables in tasks and in configuration templates, it can be used in a wide variety of infrastructure that needs Apache installed.

Ansible Galaxy

Ansible roles are powerful and flexible; they allow you to encapsulate sets of configuration and deployable units of playbooks, variables, templates, and other files, so you can easily reuse them across different servers.

It's annoying to have to start from scratch every time, though; wouldn't it be better if people could share roles for commonly-installed applications and services?

Enter Ansible Galaxy[71].

Ansible Galaxy, or just 'Galaxy', is a repository of community-contributed roles for common Ansible content. There are already hundreds of roles available which can configure and deploy common applications, and they're all available through the `ansible-galaxy` command, introduced in Ansible 1.4.2.

Galaxy offers the ability to add, download, and rate roles. With an account, you can contribute your own roles or rate others' roles (though you don't need an account to use roles).

Getting roles from Galaxy

One of the primary functions of the `ansible-galaxy` command is retrieving roles from Galaxy. Roles must be downloaded before they can be used in playbooks.

Remember the basic LAMP (Linux, Apache, MySQL and PHP) server we installed earlier in the book? Let's create it again, but this time, using a few roles from Galaxy:

```
$ ansible-galaxy install geerlingguy.apache geerlingguy.mysql geerlingg\
uy.php
```

[71]https://galaxy.ansible.com/

The latest version of a role will be downloaded if no version is specified. To specify a version, add the version after the role name, for example: `$ ansible-galaxy install geerlingguy.apache,1.0.0`.

Ansible Galaxy is still evolving rapidly, and has seen many improvements. There are a few areas where Galaxy could use some improvement (like browsing for roles by Operating System in the online interface, or automatically downloading roles that are included in playbooks), but most of these little bugs or rough spots will be fixed in time. Please check Ansible Galaxy's About[72] page and stay tuned to Ansible's blog for the latest updates.

Using role requirements files to manage dependencies

If your infrastructure configuration requires five, ten, fifteen or more Ansible roles, installing them all via `ansible-galaxy install` commands can be exhausting. Additionally, if you host roles internally (e.g. via an internal Git or Mercurial repository), you can't install the roles through Ansible Galaxy. You can, however, pass the `ansible-galaxy` command a "requirements" file with the `-r` option to automatically download all dependencies.

Ansible allows a simple `.txt` format that is very basic (though this format is deprecated and may be removed), but you should use the more standard and expressive YAML format, which allows you to install roles from Ansible Galaxy, GitHub, an HTTP download, BitBucket, or your own repository. It also allows you to specify the path into which the roles should be downloaded. An example `requirements.yml` file looks like this:

[72]https://galaxy.ansible.com/intro

```
---
# From Ansible Galaxy, latest version.
- src: geerlingguy.firewall

# From Ansible Galaxy, specifying the version.
- src: geerlingguy.php
  version: 3.5.1

# From GitHub, with a custom name and version.
- src: https://github.com/geerlingguy/ansible-role-passenger
  name: passenger
  version: 1.2.0

# From a web server, with a custom name.
- src: https://www.example.com/ansible/roles/my-role-name.tar.gz
  name: my-role
```

To install the roles defined in a requirements file, use the command `ansible-galaxy install -r requirements.yml`. For more documentation on Ansible requirements files, see the official documentation: Installing Multiple Roles From a File[73].

A LAMP server in nine lines of YAML

With the Apache, MySQL, and PHP roles installed, we can quickly create a LAMP server. This example assumes you already have an Ubuntu-based linux VM or server booted and can connect to it or run Ansible as a provisioner via Vagrant on it, and that you've run the `ansible-galaxy install` command above to download the required roles.

First, create an Ansible playbook named `lamp.yml` with the following contents:

[73]https://galaxy.ansible.com/docs/using/installing.html#installing-multiple-roles-from-a-file

```
---
- hosts: all
  become: yes

  roles:
    - geerlingguy.mysql
    - geerlingguy.apache
    - geerlingguy.php
    - geerlingguy.php-mysql
```

Now, run the playbook against a host:

```
$ ansible-playbook -i path/to/custom-inventory lamp.yml
```

After a few minutes, an entire LAMP server should be set up and running. If you add in a few variables, you can configure virtualhosts, PHP configuration options, MySQL server settings, etc.

On RHEL servers, you should add the role `geerlingguy.repo-epel` to the `roles` list after installing it via `ansible-galaxy`, because some of the required PHP packages are only available in EPEL[74].

We've effectively reduced about thirty lines of YAML (from previous examples dealing with LAMP or LAMP-like servers) down to four. Obviously, the roles have extra code in them, but the power here is in abstraction. Since most companies have many servers using similar software, but with slightly different configurations, having centralized, flexible roles saves a lot of repetition.

You could think of Galaxy roles as glorified packages; they not only install software, but they configure it *exactly* how you want it, every time, with minimal adjustment. Additionally, many of these roles work across different flavors of Linux and UNIX, so you have better configuration portability!

[74]https://fedoraproject.org/wiki/EPEL

A Solr server in seven lines of YAML

Let's grab a couple more roles and build an Apache Solr search server, which requires Java to be installed and configured.

```
$ ansible-galaxy install geerlingguy.java geerlingguy.solr
```

Then create a playbook named `solr.yml` with the following contents:

```
---
- hosts: all
  become: yes

  roles:
    - geerlingguy.java
    - geerlingguy.solr
```

Now we have a fully-functional Solr server, and we could add some variables to configure it exactly how we want, by using a non-default port, or changing the memory allocation for Solr.

A role's page on the Ansible Galaxy website highlights available variables for setting things like what version of Solr to install, where to install it, etc. For an example, view the geerlingguy.solr Galaxy page[75].

You can build a wide variety of servers with minimal effort with existing contributed roles on Galaxy. Instead of having to maintain lengthy playbooks and roles unique to each server, Galaxy lets you build a list of the required roles, and a few variables that set up the servers with the proper versions and paths. Configuration management with Ansible Galaxy becomes *true* configuration management—you get to spend more time managing your server's configuration, and less time on packaging and building individual services!

[75]https://galaxy.ansible.com/geerlingguy/solr/

Helpful Galaxy commands

Some other helpful `ansible-galaxy` commands you might use from time to time:

- `ansible-galaxy list` displays a list of installed roles, with version numbers
- `ansible-galaxy remove [role]` removes an installed role
- `ansible-galaxy init` can be used to create a role template suitable for submission to Ansible Galaxy

You can configure the default path where Ansible roles will be downloaded by editing your `ansible.cfg` configuration file (normally located in `/etc/ansible/ansible.cfg`), and setting a `roles_path` in the `[defaults]` section.

Contributing to Ansible Galaxy

If you've been working on some useful Ansible roles, and you'd like to share them with others, all you need to do is make sure they follow Ansible Galaxy's basic template (especially within the `meta/main.yml` and `README.md` files). To get started, use `ansible-galaxy init` to generate a basic Galaxy template, and make your own role match the Galaxy template's structure.

Then push your role up to a new project on GitHub (I usually name my Galaxy roles like `ansible-role-[rolename]`, so I can easily see them when browsing my repos on GitHub), and add a new role while logged into galaxy.ansible.com, under the 'My Content' tab.

Summary

Using includes and Ansible roles organizes Playbooks and makes them maintainable. This chapter introduced different ways of including tasks, playbooks, and handlers, the power and flexible structure of roles, and how you can utilize Ansible Galaxy, the community repository of configurable Ansible roles that do just about anything.

```
 _________________________________________
/ When the only tool you own is a hammer, \
| every problem begins to resemble a      |
\ nail. (Abraham Maslow)                  /
 -----------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```