Ansible is a configuration management and provisioning tool, similar to Chef, Puppet or Salt.

I've found it to be one of the simplest and the easiest to get started with. A lot of this is because it's "just SSH"; It uses SSH to connect to servers and run the configured Tasks.

One nice thing about Ansible is that it's very easy to convert bash scripts (still a popular way to "do" configuration management) into Ansible Tasks. Since it's primarily SSH based, it's not hard to see why this might be the case - Ansible ends up running the same (ish) commands.

We could just script our own provisioners, but Ansible is much cleaner because it automates the process of getting *context* before running Tasks. With this context, Ansible is able to handle most edge cases - the kind we usually take care of with long, increasingly complex scripts.

Ansible Tasks are idempotent. Without a lot of extra coding, bash scripts are usually **not** safety run again and again. Ansible uses "Facts", which is system and environment information it gathers ("context") before running Tasks.

Ansible uses these facts to check state and see if it needs to change anything in order to get the desired outcome. This makes it safe to run Ansible Tasks against a server over and over again.

Here I'll show how easy it is to get started with Anible. We'll start basic and then add in more features as we improve upon our configurations.

Install

Of course we need to start by installing Ansible. Tasks can be run off of any machine Ansible is installed on.

This means there's usually a "central" server running Ansible commands, although there's nothing particularly special about what server Ansible is installed on. Ansible is "agentless" - there's no central agent(s) running. We can even run Ansible from any server; I often run Tasks from my laptop.

Ubuntu

Here's how to install Ansible on Ubuntu 16.04. We'll use the easy-to-remember `ppa:ansible/ansible` repository as per the official docs.

```
sudo apt-add-repository -y ppa:ansible/ansible
sudo apt-get update
sudo apt-get install -y ansible
```

This installs Ansible globally.

Virtual env's

This is my preferred way to install Ansible.

We can also use Pip to install virtualenv, which lets us install Python libraries in their own little environment that won't affect others (nor force us to install tools globally).

Here's how:

```
# Install python2.7 (Ubuntu 16.04 comes with python 3 out of the box) and
Pip
sudo apt-get install -y python2.7 python-pip

## Use Pip to install virtualenv
### -U updates it if the package is already installed
sudo pip install -U virtualenv
```

Once we have pip and virtualenv installed globally, we can get Ansible inside of a virtual environment:

```
# Go to my user's home directory,
# make a directory to play with ansible
cd ~/
mkdir ansible-play
cd ansible-play

# Create a python virtual environment
virtualenv .venv
# Enable the virtual environment
source .venv/bin/activate

# Then anything we intall with pip will be
# inside that virtual environment
pip install ansible
```

Those commands will install the latest stable Ansible 2 (as of this writing).

Later, when you're done, you can deactivate the virtualenv via the `deactivate` command

At any time, you can update ansible by running:

```
# Assumes the virtualenv is active - `source .venv/bin/activate`
# Assuming the virtualenv is active
pip install -U ansible
```

Configuration

A common use is to put Ansible in a virtualenv as we've done above. In that case, we won't have (or want!) those default files. We'll continue on this way - we can create the configuration files within our local directory as needed. We won't need any configuration files in `/etc` or other locations, which makes it a little more sane to use.

Managing Servers: Inventory

Ansible has you create an inventory file used to define which servers it will be managing. This file can be named anything, but we'll typically name it `hosts`.

Within the `hosts` file, we can define some servers to manage. Here's we'll define two servers we may want to manage under the "web" label. The label is arbitrary:

```
[web]
192.168.22.10
192.168.22.11
```

That's good enough for now. If needed, we can define ranges of hosts, multiple groups, reusable variables, and use other fancy setups, including creating a dynamic inventory.

For testing this article, I spun up an Ubuntu 16.04 server on AWS (doesn't matter where or what cloud you use though). Then I then ran Ansible Tasks directly on that server. This means I'm running Ansible on the same server that I'm managing, which is valid way to run Ansible (although not it's main use case).

When we run Ansible against the local machine, we don't need to care about what's in the inventory file - I'll show you running Ansible locally and against a remote server.

For now, let's set the `hosts` file to point to local host under `local` and a fake remote host under the `remote` name:

```
[local]
127.0.0.1

[remote]
192.168.1.2
```

I'll show you commands for running connects against localhost and remote servers.

Basics: Running Commands

Let's start running Tasks against a server.

Ansible will assume you have SSH access available to your servers, usually based on SSH-Key. Because Ansible uses SSH, the server it's on needs to be able to SSH into the inventory servers.

However, Ansible will attempt to connect as the current user it is being run as. If I'm running Ansible as user `ubuntu` (as is the case for me on AWS), it will attempt to connect as user `ubuntu` on the other servers.

If Ansible can directly SSH into the managed servers, we can run commands without too much fuss:

```
# Run against localhost
$ ansible -i ./hosts --connection=local local -m ping

# Run against remote server
$ ansible -i ./hosts remote -m ping
127.0.0.1 | success >> {
    "changed": false,
    "ping": "pong"
```

```
}
```

If you get an SSH error here for "Too many authentication failures", we can add some SSH options, and have Ansible ask us for our password to log in with: `ansible -i ./hosts --ask-pass --ssh-extra-args='-o "PubkeyAuthentication=no"' all -m ping`.

Using `--connection=local` tells Ansible to not attempt to run the commands over SSH, since we're just affecting the local host. However, we still need a `hosts` file telling us where to connect to - it won't assume `localhost` or `127.0.0.1` by itself (for some reason!).

In either case, we can see the output we get from Ansible is some JSON which tells us if the Task (our call to the `ping` module) made any changes and the result.

Let's cover these commands:

- `-i ./hosts` - Set the inventory file, the one named `hosts`
- `remote`, `local`, `all` - Use the servers defined under this label in the `hosts` inventory file. "all" is a special keyword to run against every server defined in the file
- `-m ping` - Use the "ping" module, which simply runs the `ping` command and returns the results
- `-c local` | `--connection=local` - Run commands on the local server, not over SSH

Modules

Ansible uses "modules" to accomplish most of its Tasks. Modules can do things like install software, copy files, use templates and much more.

Modules are *the* way to use Ansible, as they can use available context ("Facts") in order to determine what actions, if any need to be done to accomplish a Task.

If we didn't have modules, we'd be left running arbitrary shell commands, and we might as well just use bash script. Here's what an arbitrary shell command looks like in Ansible (it's using the `shell` module!):

```
# Run against a local server
ansible -i ./hosts local --connection=local -b --become-user=root \
    -m shell -a 'apt-get install nginx'

# Run against a remote server
ansible -i ./hosts remote -b --become-user=root all \
    -m shell -a 'apt-get install nginx'
```

Here, the `sudo apt-get install nginx` command will be run using the "shell" module.

We have some new flags too:

- `-b` - "become", tell Ansible to become another user when running the command. This is how you run as different users or promote yourself to the `root` user.
- `--become-user=root` - Run the following commands as user "root" (e.g. use "sudo" with the command). We can define any existing user here.
    - `-a` used to pass any arguments to the module defined with `-m`

However this isn't particularly powerful. While it's handy to be able to run these commands on all of our servers at once, we still only accomplish what any bash script might do.

If we used a more appropriate module instead, we can run commands with an assurance of the result. Ansible modules ensure indempotence - we can run the same Tasks over and over without affecting the final result.

For installing software on Debian/Ubuntu servers, the "apt" module will run the same command, but ensure idempotence.

```
# Run against a local server
ansible -i ./hosts local --connection=local -b --become-user=root \
    -m apt -a 'name=nginx state=installed update_cache=true'

127.0.0.1 | success >> {
    "changed": false
}

# Run against a remote server
ansible -i ./hosts remote -b --become-user=root \
    -m apt -a 'name=nginx state=installed update_cache=true'

127.0.0.1 | success >> {
    "changed": false
}
```

This will use the apt module to update the repository cache and install Nginx (if not installed).

The result of running the Task was `"changed": false`. This shows that there were no changes; I had already installed Nginx using the `shell` module. The nice thing is that I can run this command over and over without worrying about it changing the desired result - Nginx is already installed, Ansible knows that, and doesn't attempt to re-install it.

Going over the command:

- `-i ./hosts` - Set the inventory file, the one named `hosts`
- `-b` - "become", tell Ansible to become another user to run the command
- `--become-user=root` - Run the following commands as user "root" (e.g. use "sudo" with the command)
- `local` | `remote` - Run on local or remote defined hosts from the inventory file
- `-m apt` - Use the apt module
- `-a 'name=nginx state=installed update_cache=true'` - Provide the arguments for the apt module, including the package name, our desired end state and whether to update the package repository cache or not

We can run all of our needed Tasks (via modules) in this ad-hoc way, but let's make this more managable. We'll move this Task into a Playbook, which can run and coordinate multiple Tasks.

Playbooks

Playbooks can run multiple Tasks and provide some more advanced functionality that we would miss out on using ad-hoc commands. Let's move the above Task into a playbook.

Playbooks and Roles in Ansible all use Yaml.

Create file `nginx.yml`:

```
---
# hosts could have been "remote" or "all" as well
- hosts: local
  connection: local
  become: yes
  become_user: root
  tasks:
   - name: Install Nginx
     apt:
       name: nginx
       state: installed
       update_cache: true
```

This Task does exactly the same as our ad-hoc command, including setting the use of a local connection.

This would use the servers under the `[local]` label in the `hosts` file.

If we weren't using a local connection, we'd do something like this:

```
---
- hosts: remote
  become: yes
  become_user: root
  tasks:
   - name: Install Nginx
     apt:
       name: nginx
       state: installed
       update_cache: true
```

This would use the servers under the `[remote]` label in the `hosts` file.

Use use `become` and `become_user` again in our Tasks file to tell Ansible to use `sudo` to run commands as user `root`, and then pass the Playbook file.

With a Yaml playbook file, we need to use the `ansible-playbook` command, which becomes simpler to run now:

```
$ ansible-playbook -i ./hosts nginx.yml

PLAY [local]
*********************************************************************
```

```
GATHERING FACTS
*************************************************************
ok: [127.0.0.1]

TASK: [Install Nginx]
*********************************************************
ok: [127.0.0.1]

PLAY RECAP
********************************************************************
127.0.0.1                  : ok=2    changed=0    unreachable=0    failed=0
```

We get some useful feedback while this runs, including the Tasks Ansible runs and their result. Here we see all ran OK, but nothing was changed. I have Nginx installed already.

Handlers

A Handler is exactly the same as a Task (it can do anything a Task can), but it will only run when called by another Task. You can think of it as part of an Event system; A Handler will take an action when called by an event it listens for.

This is useful for "secondary" actions that might be required after running a Task, such as starting a new service after installation or reloading a service after a configuration change.

```
---
# Example shows using the local machine still
# Remove 'connection' and set hosts to 'remote' for a remote connection
- hosts: local
  connection: local
  become: yes
  become_user: root
  tasks:
   - name: Install Nginx
     apt:
       name: nginx
       state: installed
       update_cache: true
     notify:
      - Start Nginx

  handlers:
   - name: Start Nginx
     service:
       name: nginx
       state: started
```

Here we add a `notify` directive to the installation Task. This notifies any Handler named "Start Nginx" after the Task is run.

Then we can create the Handler called "Start Nginx". This Handler is the Task called when "Start Nginx" is notified.

This particular Handler uses the Service module, which can start, stop, restart, reload (and so on) system services. In this case, we tell Ansible that we want Nginx to be started.

Note that Ansible has us define the ***state*** you wish the service to be in, rather than defining the ***change*** you want. Ansible will decide if a change is needed, we just tell it the desired result.

Let's run this Playbook again:

```
$ ansible-playbook -i ./hosts nginx.yml

PLAY [local]
*********************************************************************

GATHERING FACTS
*********************************************************************
ok: [127.0.0.1]

TASK: [Install Nginx]
*****************************************************
ok: [127.0.0.1]

NOTIFIED: [nginx | Start Nginx]
*******************************************
ok: [127.0.0.1]

PLAY RECAP
********************************************************************
127.0.0.1                  : ok=2    changed=0    unreachable=0    failed=0
```

We get the similar output, but this time the Handler was run.

Notifiers are only run if the Task is run. If I already had Nginx installed, the Install Nginx Task would not be run and the notifier would not be called.

We can use Playbooks to run multiple Tasks, add in variables, define other settings and even include other playbooks.

More Tasks

Next we can add a few more Tasks to this Playbook and explore some other functionality.

```
---
# Example shows using the local machine still
# Remove 'connection' and set hosts to 'remote' for a remote connection
- hosts: local
  connection: local
  become: yes
  become_user: root
  vars:
   - docroot: /var/www/serversforhackers.com/public
  tasks:
   - name: Add Nginx Repository
     apt_repository:
       repo: ppa:nginx/stable
       state: present
     register: ppastable

   - name: Install Nginx
     apt:
```

```
      pkg: nginx
      state: installed
      update_cache: true
    when: ppastable|success
    notify:
     - Start Nginx

  - name: Create Web Root
    file:
     path: '{{ docroot }}'
     mode: 775
     state: directory
     owner: www-data
     group: www-data
    notify:
     - Reload Nginx

handlers:
 - name: Start Nginx
   service:
     name: nginx
     state: started

  - name: Reload Nginx
    service:
      name: nginx
      state: reloaded
```

There are now three Tasks:

- `Add Nginx Repository` - Add the Nginx stable PPA to get the latest stable version of Nginx, using the apt_repository module.
- `Install Nginx` - Installs Nginx using the Apt module.
- `Create Web Root` - Finally, create a web root directory.

Also new here are the `register` and `when` directives. These tell Ansible to run a Task **when** something else happens.

The "Add Nginx Repository" Task registers "ppastable". Then we use that to inform the Install Nginx Task to only run when the registered "ppastable" Task is successful. This allows us to conditionally stop Ansible from running a Task.

This particular example of only installing Nginx when the ppa repository was added is superfluous, as if adding the repository fails, Ansible will stop and report the error. However it's good to know the functionality exists.

You can register the results of a modules action as well, and use the variable defined in `register` to conditionally perform actions `when` based on the registered variables values. For example, registering the result of the command run via the `shell` module can let you access the stdout of that command.

We also use a variable. The `docroot` variable is defined in the `var` section. It's then used as the destination argument of the file module which creates the defined directory.

Note that the `path` configuration uses brackets `{{ var-name }}` - This is Jinja2 templating. In order for Ansible to parse the Jinja2 template variable within the brackets, the line must be in single or double quotes - e.g. `path: '{{ docroot }}'` instead of `path: {{ docroot }}`. Not using quotes will result in an error.

This playbook can be run with the usual command:

```
ansible-playbook -i ./hosts nginx.yml
```

So, we've run some ad-hoc commands, used Ansible modules, and organized a few related tasks into a playbook.

Next we'll take Ansible further by organizing the Playbook into a Role, which helps us organize related items such as files and templates, while also helping us organize more complex related tasks and actions.

Roles

Roles are good for organizing multiple, related Tasks and encapsulating data needed to accomplish those Tasks. For example, installing Nginx may involve adding a package repository, installing the package, and setting up configuration. We've seen installation via a Playbook, but once we start configuring our installations, the Playbooks tend to get a little more busy.

Furtermore, real-world configuration often requires extra data such as variables, files, dynamic templates and more. These tools can be used with Playbooks, but we can do better immediately by **organizing related Tasks and data into one coherent structure: a Role**.

Roles have a directory structure like this:

```
roles
  rolename
   - files
   - handlers
   - meta
   - templates
   - tasks
   - vars
```

Within each directory, Ansible will search for and read any Yaml file called `main.yml` automatically.

We'll break apart our `nginx.yml` file and put each component within the corresponding directory to create a cleaner and more complete provisioning toolset.

Creating a Role

We can use the `ansible-galaxy` command to create a new role. This tool can be used to save roles to Ansible's public registry, however I generally just use it to bootrap a role locally.

Let's see how to set this up:

```
# Head to our previously created directory
cd ~/ansible-example

# In case we left our virtualenv at some point
source .venv/bin/activate

# Create a roles directory
mkdir roles
cd roles

# Bootstrap a new role named "nginx"
ansible-galaxy init nginx
```

The directory name `roles` is a convention Ansible uses to find roles when running a playbook. The directory should always be named `roles`.

The `ansible-galaxy init nginx` command, run within the `roles` directory, will create the directories/files needed to get started with a new role.

Let's run through each part of our new Nginx role found at `~/ansible-example/roles/nginx`.

Files

First, within the `files` directory, we can add files that we'll want copied into our servers. For Nginx, I often copy H5BP's Nginx component configurations. I simply download the latest from Github, make any tweaks I want, and put them into the `files` directory.

```
~/ansible-example
 - roles
 - - nginx
 - - - files
 - - - - h5bp
```

As we'll see in a bit, the H5BP configuration files will be added to the server via the copy module.

Handlers

Inside of the `handlers` directory, we can put all of our Handlers that were once within the `nginx.yml` Playbook.

Inside of `handlers/main.yml`:

```
---
- name: Start Nginx
  service:
    name: nginx
    state: started

- name: Reload Nginx
  service:
```

```
    name: nginx
    state: reloaded
```

Once these are in place, we can reference them from other yaml configuration freely.

Meta

The `main.yml` file within the `meta` directory contains Role meta data, including dependencies.

If this Role depended on another Role, we could define that here. For example, I have the Nginx Role depend on the SSL Role, which installs SSL certificates.

```
---
dependencies:
  - { role: ssl }
```

If I called the "nginx" Role, it would attempt to first run the "ssl" Role.

Otherwise we can omit this file, or define the Role as having no dependencies:

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

Template

Template files can contain template variables, based on Python's Jinja2 template engine. Files in here should end in `.j2`, but can otherwise have any name. Similar to `files`, we won't find a `main.yml` file within the `templates` directory.

Here's an example Nginx server ("virtual host") configuration. Note that it uses some variables which we'll define later in the `vars/main.yml` file.

This Nginx config file in our example is placed at `templates/serversforhackers.com.conf.j2`:

```
server {
    # Enforce the use of HTTPS
    listen 80 default_server;
    server_name {{ domain }};
    return 301 https://$server_name$request_uri;
}

server {
    listen 443 ssl default_server;

    root /var/www/{{ domain }}/public;
    index index.html index.htm index.php;

    access_log /var/log/nginx/{{ domain }}.log;
    error_log  /var/log/nginx/{{ domain }}-error.log error;

    server_name {{ domain }};
```

```
    charset utf-8;

    include h5bp/basic.conf;

    ssl_certificate           {{ ssl_crt }};
    ssl_certificate_key       {{ ssl_key }};
    include h5bp/directive-only/ssl.conf;

    location / {
        try_files $uri $uri/ /index.php$is_args$args;
    }

    location = /favicon.ico { log_not_found off; access_log off; }
    location = /robots.txt  { log_not_found off; access_log off; }

    location ~ \.php$ {
        include snippets/fastcgi.conf;
        fastcgi_pass unix:/var/run/php7.1-fpm.sock;
    }
}
```

This is a fairly standard Nginx configuration for a PHP application. There are three variables used here:

- domain
- ssl_crt
- ssl_key

These three variables will be defined in the variables section.

Variables

Before we integrate everything together using the Tasks, let's look at variables. The `vars` directory contains a `main.yml` file which simply lists variables we'll use. This provides a convenient place for us to change configuration-wide settings.

Here's what the `vars/main.yml` file might look like:

```
---
domain: serversforhackers.com
ssl_key: /etc/ssl/sfh/sfh.key
ssl_crt: /etc/ssl/sfh/sfh.crt
```

These are three variables which we can use elsewhere in this Role. We saw them used in the template above, but we'll see them in our defined Tasks as well.

If you have sensitive information to add into a variable file, you can encrypt the file using `ansible-vault`, which is explained more below.

Tasks

Let's finally see this all put together into a series of Tasks.

The main file run when we use a role is the `tasks/main.yml` file. Let's see what that will look like for our use case:

```
---
- name: Add Nginx Repository
  apt_repository:
    repo: ppa:nginx/stable
    state: present

- name: Install Nginx
  apt:
    pkg: nginx
    state: installed
    update_cache: true
  notify:
    - Start Nginx

- name: Add H5BP Config
  copy:
    src: h5bp
    dest: /etc/nginx
    owner: root
    group: root

- name: Disable Default Site Configuration
  file:
    dest: /etc/nginx/sites-enabled/default
    state: absent

# `dest` in quotes as a variable is used!
- name: Add SFH Site Config
  register: sfhconfig
  template:
    src: serversforhackers.com.j2
    dest: '/etc/nginx/sites-available/{{ domain }}.conf'
    owner: root
    group: root

# `src`/`dest` in quotes as a variable is used!
- name: Enable SFH Site Config
  file:
    src: '/etc/nginx/sites-available/{{ domain }}.conf'
    dest: '/etc/nginx/sites-enabled/{{ domain }}.conf'
    state: link

# `dest` in quotes as a variable is used!
- name: Create Web root
  file:
    dest: '/var/www/{{ domain }}/public'
    mode: 775
    state: directory
    owner: www-data
    group: www-data
  notify:
    - Reload Nginx

# `dest` in quotes as a variable is used!
- name: Web Root Permissions
  file:
   dest: '/var/www/{{ domain }}'
```

```
    mode: 775
    state: directory
    owner: www-data
    group: www-data
    recurse: yes
  notify:
    - Reload Nginx
```

This is a longer series of Tasks, which makes for a more complete installation of Nginx. The Tasks, in order of appearance, accomplish the following:

- Add the nginx/stable repository
- Install & start Nginx
- Add H5BP configuration files
- Disable the default Nginx configuration by removing the symlink to the `default` file from the `sites-enabled` directory
- Copy the `serversforhackers.com.conf.j2` virtual host template into the Nginx configuration, rendering the template as it does
- Enable the Nginx server configuration by symlinking it to the `sites-enabled` directory
- Create the web root directory
- Change permission (recursively) for the project root directory, which is one level above the web root created previously

There's some new modules (and new uses of some we've covered), including **copy**, **template**, & **file** modules. By setting the arguments for each module, we can do some interesting things such as ensuring files are "absent" (delete them if they exist) via `state: absent`, or create a file as a symlink via `state: link`. You should check the docs for each module to see what interesting and useful things you can accomplish with them.

Running the Role

To run one or more roles against a server, we'll re-use another playbook. The playbook should be in the same directory as the `roles` directory, which is where we need to `cd` into as well when running the `ansible-playbook` command.

Remove the `ssl` dependency from `meta/main.yml` before running this Role if you are following along and added that.

Let's create a "master" yaml file which defines the Roles to use and what hosts to run them on:

File `~/ansible-example/server.yml`, which is in the same directory as the `roles` directory:

```
---
# run locally here, yadda yadda yadda
- hosts: local
  connection: local
  roles:
    - nginx
```

So, instead of defining all our variables and tasks in this Playbook file, we simply define the Role. The Role takes care of the gritty details.

Then we can run the Role(s):

```
ansible-playbook -i ./hosts server.yml
```

Here's the output from the run of the Playbook file, which runs the Nginx Role:

```
PLAY [all]
*********************************************************************

GATHERING FACTS
*********************************************************************
ok: [127.0.0.1]

TASK: [nginx | Add Nginx Repository]
***************************************
changed: [127.0.0.1]

TASK: [nginx | Install Nginx]
***********************************************
changed: [127.0.0.1]

TASK: [nginx | Add H5BP Config]
*********************************************
changed: [127.0.0.1]

TASK: [nginx | Disable Default Site]
***************************************
changed: [127.0.0.1]

TASK: [nginx | Add SFH Site Config]
*****************************************
changed: [127.0.0.1]

TASK: [nginx | Enable SFH Site Config]
***************************************
changed: [127.0.0.1]

TASK: [nginx | Create Web root]
*******************************************
changed: [127.0.0.1]

TASK: [nginx | Web Root Permissions]
***************************************
ok: [127.0.0.1]

NOTIFIED: [nginx | Start Nginx]
*********************************************
ok: [127.0.0.1]

NOTIFIED: [nginx | Reload Nginx]
*******************************************
changed: [127.0.0.1]

PLAY RECAP
*********************************************************************
127.0.0.1                  : ok=8   changed=7   unreachable=0    failed=0
```

Awesome, we put all the various components together into a coherent Role and now have Nginx installed and configured!

Facts

Notice that the first line when running a playbook is always "gathering facts".

Before running any Tasks, Ansible will gather information about the system it's provisioning. These are called Facts, and include a wide array of system information such as the number of CPU cores, available ipv4 and ipv6 networks, mounted disks, Linux distribution and much more.

Facts are often useful in Tasks or Template configurations. For example Nginx is commonly set to use as many worker processors as there are CPU cores. Knowing this, you may choose to set your template of the `nginx.conf.j2` file like so:

```
user www-data;
worker_processes {{ ansible_processor_cores }};
pid /var/run/nginx.pid;

# And other configurations...
```

Or if you have a server with multiple CPU's, you can use:

```
user www-data;
worker_processes {{ ansible_processor_cores * ansible_processor_count }};
pid /var/run/nginx.pid;

# And other configurations...
```

Ansible facts all start with `anisble_` and are globally available for use any place variables can be used: Variable files, Tasks, and Templates.

Try running the following against your local machine to see what facts are available:

```
# Run against a local server
# Note that we say to use "localhost" instead of defining a hosts file
here!
ansible -m setup --connection=local localhost

# Run against a remote server
ansible -i ./hosts remote -m setup
```

Vault

We often need to store sensitive data in our Ansible templates, Files or Variable files; It unfortunately cannot always be avoided (which is a pain when we check these files into a remote Git repository). Ansible has a solution for this called Ansible Vault.

Vault allows you to encrypt any Yaml file, which typically boil down to our Variable files. Vault will **not** encrypt Files and Templates, only Yaml files.

When creating an encrypted file, you'll be asked a password which you must use to edit the file later and when calling the Roles or Playbooks.

Keep the password you create somewhere safe.

For example we can create a new Variable file:

```
ansible-vault create vars/main.yml
Vault Password:
```

After entering in the encryption password, the file will be opened in your default editor, usually Vim or Nano.

The editor used is defined by the `EDITOR` environmental variable. The default is usually Vim. If you are not a Vim user, you can change it quickly by setting the environmental variabls:

```
EDITOR=nano ansible-vault edit vars/main.yml
```

T> The editor can be set in the users profile/bash configuration, usually found at `~/.profile`, `~/.bashrc`, `~/.zshrc` or similar, depending on the shell and Linux distribution used.

Ansible Vault itself is fairly self-explanatory. Here are the commands you can use:

```
$ ansible-vault -h
Usage: ansible-vault [create|decrypt|edit|encrypt|rekey] \
      [--help] [options] file_name

Options:
  -h, --help  show this help message and exit
```

For the most part, we'll use `ansible-vault create|edit /path/to/file.yml`. Here, however, are all of the available commands:

- **create** - Create a new file and encrypt it
- **decrypt** - Create a plaintext file from an encrypted file
- **edit** - Edit an already-existing encrypted file
- **encrypt** - Encrypt an existing plain-text file
- **rekey** - Set a new password on a encrypted file

If you have an existing configuration file to encrypt, use `ansible-vault encrypt /path/to/file.yml`.

Example: Users

Let's pretend we have a second Role named "users":

```
cd ~/ansible-example/roles
ansible-galaxy init users
```

I use Vault when creating new users and setting their passwords. In a User Role, you can set a Variable file with users' passwords and a public key to add to the users' `authorized_keys` file (thus giving you SSH access).

T> Public SSH keys are technically safe for the general public to see - all someone can do with them is allow you access to their own servers. Public keys are intentionally useless for gaining access to a sytem without the paired private key, which we are not putting into this Role.

Here's an example variable file which can be created and encrypt with Vault. While editing it, it's of course in plain-text.

Here is file `~/ansible-example/roles/users/variables/main.yml`:

```
admin_password: $6$lpQ1DqjZQ25gq9YW$mHZAmGhFpPVVv0JCYUFaDovu8u5EqvQi.Ih
deploy_password: $6$edOqVumZrYW9$d5zj1Ok/G80DrnckixhkQDpXl0fACDfNx2EHnC
common_public_key: ssh-rsa ALongSSHPublicKeyHere
```

Note that the passwords for the users are also hashed. You can read Ansible's documentation on generating encrypted passwords, which the User module requires to set a user password. As a quick primer, it looks like this on Ubuntu:

```
# The whois package makes the mkpasswd
# command available on Ubuntu
$ sudo apt-get install -y whois

# Create a password hash
$ mkpasswd --method=SHA-512
Password:
```

This will generate a hashed password for you to use with the `user` module.

The passwords in the variable file are hashed, but I still like encrypting the yaml file containing the hashed passwords. Often these files contain unhashed data such as API tokens or SSH private keys, making encryption very important.

Once you have set the user passwords and added the public key into the Variables file we can encrypt the file and then make a Task to use these encrypted variables.

```
ansible-vault encrypt roles/users/variables/main.yml
> INPUT PASSWORD
```

Then we can edit our tasks file to use add our new users with the (encrypted) variables:

Here is file `~/ansible-example/roles/users/tasks/main.yml`:

```
---
- name: Create Admin User
  user:
    name: admin
    password: '{{ admin_password }}'
    groups: sudo
    append: yes
```

```
    shell: /bin/bash

- name: Add Admin Authorized Key
  authorized_key:
    user: admin
    key: '{{ common_public_key }}'
    state: present

- name: Create Deploy User
  user:
    name: deploy
    password: '{{ deploy_password }}'
    groups: www-data
    append: yes
    shell: /bin/bash

- name: Add Deployer Authorized Key
  authorized_key:
    user: deploy
    key: '{{ common_public_key }}'
    state: present
```

These Tasks use the `user` module to create new users, passing in the passwords set in the Variable file.

It also uses the `authorized_key` module to add the SSH pulic key as an authorized SSH key in the server for each user.

Encrypted variables are used like usual within the Tasks file. However, in order to run this Role, we'll need to tell Ansible to ask for the Vault password so it can unencrypt the variables.

Let's edit our `server.yml` Playbook file to call our `user` Role:

```
---
# Local connection here, yadda yadda yadda
- hosts: local
  connection: local
  sudo: yes
  roles:
    - nginx
    - user
```

To run this Playbook, we need to tell Ansible to ask for the Vault password, as we're running a Role which contains an encrypted file:

```
ansible-playbook --ask-vault-pass -i ./hosts server.yml
```

Recap

Here's what we did:

- Installed Ansible
- Configured an Ansible inventory file (only technically needed when not using `connection: local`)

- Ran idempotent ad-hoc commands on multiple servers simultaneously
- Created a basic Playbook to run multiple Tasks, using Handlers
- Abstracted the Tasks into a Role to keep everything Nginx-related organized
    - Saw how to set dependencies
    - Saw how to register Task "dependencies" and run other Tasks only if they are successful
    - Saw how to use more templates, files and variables in our Tasks
- Saw how to incorporate Ansible Facts
- Saw how to use Ansible Vault to add security to our variables

We covered a lot of ground, and it will get you pretty far. However, there are many more tools provided by Ansible to explore!

Additional Resources

- Use Ansible Vault to encrypt secure data, making it more safe if your playbooks or Roles end up in version control or are otherwise shared
- See the Module Index for more information on available modules

There are a **lot** of useful things you can do with Ansible. Explore the documentation!
ommon_public_key }}" state=present

```
These Tasks use the `user` module to create new users, passing in the
passwords set in the Variable file.

It also uses the `authorized_key` module to add the SSH pulic key as an
authorized SSH key in the server for each user.

Variables are used like usual within the Tasks file. However, in order to
run this Role, we'll need to tell Ansible to ask for the Vault password so
it can unencrypt the variables.

Let's setup a `provision.yml` Playbook file to call our `user` Role:

```yaml
---
- hosts: all
  sudo: yes
  roles:
    - user
```

To run this Playbook, we need to tell Ansible to ask for the Vault password, as we're running a Role which contains an encrypted file:

```
ansible-playbook --ask-vault-pass provision.yml
```
```