Chapter 4 - Ansible Playbooks

Power plays

Like many other configuration management solutions, Ansible uses a metaphor to describe its configuration files. They are called 'playbooks', and they list sets of tasks ('plays' in Ansible parlance) that will be run against a particular server or set of servers. In American football, a team follows a set of pre-written playbooks as the basis for a bunch of plays they execute to try to win a game. In Ansible, you write playbooks (a list of instructions describing the steps to bring your server to a certain configuration state) that are then *play*ed on your servers.

Playbooks are written in YAML[50], a simple human-readable syntax popular for defining configuration. Playbooks may be included within other playbooks, and certain metadata and options cause different plays or playbooks to be run in different scenarios on different servers.

Ad-hoc commands alone make Ansible a powerful tool; playbooks turn Ansible into a top-notch server provisioning and configuration management tool.

What attracts most DevOps personnel to Ansible is the fact that it is easy to convert shell scripts (or one-off shell commands) directly into Ansible plays. Consider the following script, which installs Apache on a RHEL/CentOS server:

Shell Script

[50]https://docs.ansible.com/ansible/latest/reference_appendices/YAMLSyntax.html

```
# Install Apache.
yum install --quiet -y httpd httpd-devel
# Copy configuration files.
cp httpd.conf /etc/httpd/conf/httpd.conf
cp httpd-vhosts.conf /etc/httpd/conf/httpd-vhosts.conf
# Start Apache and configure it to run at boot.
service httpd start
chkconfig httpd on
```

To run the shell script (in this case, a file named `shell-script.sh` with the contents as above), you would call it directly from the command line:

```
# (From the same directory in which the shell script resides).
$ ./shell-script.sh
```

Ansible Playbook

```
---
- hosts: all

  tasks:
    - name: Install Apache.
      command: yum install --quiet -y httpd httpd-devel
    - name: Copy configuration files.
      command: >
        cp httpd.conf /etc/httpd/conf/httpd.conf
    - command: >
        cp httpd-vhosts.conf /etc/httpd/conf/httpd-vhosts.conf
    - name: Start Apache and configure it to run at boot.
      command: service httpd start
    - command: chkconfig httpd on
```

To run the Ansible Playbook (in this case, a file named `playbook.yml` with the contents as above), you would call it using the `ansible-playbook` command:

```
# (From the same directory in which the playbook resides).
$ ansible-playbook playbook.yml
```

Ansible is powerful in that you quickly transition to using playbooks if you know how to write standard shell commands—the same commands you've been using for years—and then as you get time, rebuild your configuration to take advantage of Ansible's helpful features.

In the above playbook, we use Ansible's `command` module to run standard shell commands. We're also giving each task a 'name', so when we run the playbook, the task has human-readable output on the screen or in the logs. The command module has some other tricks up its sleeve (which we'll see later), but for now, be assured shell scripts are translated directly into Ansible playbooks without much hassle.

> The greater-than sign (>) immediately following the `command:` module directive tells YAML "automatically quote the next set of indented lines as one long string, with each line separated by a space". It helps improve task readability in some cases. There are different ways of describing configuration using valid YAML syntax, and these methods are discussed in-depth in the YAML Conventions and Best Practices section in Appendix B.
>
> This book uses three different task-formatting techniques: For tasks which require one or two simple parameters, Ansible's shorthand syntax (e.g. `yum: name=apache2 state=present`) is used. For most uses of `command` or `shell`, where longer commands are entered, the > technique mentioned above is used. For tasks which require many parameters, YAML object notation is used—placing each key and variable on its own line. This assists with readability and allows for version control systems to easily distinguish changes line-by-line.

The above playbook will perform *exactly* like the shell script, but you can improve things greatly by using some of Ansible's built-in modules to handle the heavy lifting:

Revised Ansible Playbook - Now with idempotence!

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

  tasks:
    - name: Install Apache.
      yum:
        name:
          - httpd
          - httpd-devel
        state: present
    - name: Copy configuration files.
      copy:
        src: "{{ item.src }}"
        dest: "{{ item.dest }}"
        owner: root
        group: root
        mode: 0644
      with_items:
        - src: httpd.conf
          dest: /etc/httpd/conf/httpd.conf
        - src: httpd-vhosts.conf
          dest: /etc/httpd/conf/httpd-vhosts.conf
    - name: Make sure Apache is started now and at boot.
      service: name=httpd state=started enabled=yes
```

Now we're getting somewhere. Let me walk you through this simple playbook:

1. The first line, `---`, is how we mark this document as using YAML syntax (like using `<html>` at the top of an HTML document, or `<?php` at the top of a block of PHP code).
2. The second line, `- hosts: all` defines the first (and in this case, only) *play*, and tells Ansible to run the play on `all` hosts that it knows about.
3. The third line, `become: yes` tells Ansible to run all the commands through `sudo`, so the commands will be run as the root user.

4. The fifth line, `tasks:`, tells Ansible that what follows is a list of tasks to run as part of this playbook.
5. The first task begins with `name: Install Apache..` `name` is not a module that does something to your server; rather, it's a way of giving a human-readable description to the task that follows. Seeing "Install Apache" is more relevant than seeing "yum name=httpd state=present"… but if you drop the name line completely, that won't cause any problem.
    - We use the `yum` module to install Apache. Instead of the command `yum -y install httpd httpd-devel`, we can describe to Ansible exactly what we want. Ansible will take the list of packages we provide. We tell yum to make sure the packages are installed with `state: present`, but we could also use `state: latest` to ensure the latest version is installed, or `state: absent` to make sure the packages are *not* installed.
6. The second task again starts with a human-readable name (which could be left out if you'd like).
    - We use the `copy` module to copy files from a source (on our local workstation) to a destination (the server being managed). We could also pass in more variables, like file metadata including ownership and permissions (`owner`, `group`, and `mode`).
    - Ansible allows lists of variables to be passed into tasks using `with_items`: Define a list of items and each one will be passed into the play, referenced using the `item` variable (e.g. `{{ item }}`).
    - In this case, we are using a list of items containing dicts (dictionaries) used for variable substitution; to define each element in a list of dicts with each list item in the format:

      ```
      - var1: value
        var2: value
      ```

      The list can have as many variables as you want, even deeply-nested dicts. When you reference the variables in the play, you use a dot to access the variable within the item, so `{{ item.var1 }}` would access the first variable. In our example, `item.src` accesses the `src` in each item.
7. The third task also uses a name to describe it in a human-readable format.
    - We use the `service` module to describe the desired state of a particular service, in this case `httpd`, Apache's http daemon. We want it to be running, so we set `state=started`, and we want it to run at system startup, so we say `enabled=yes` (the equivalent of running `chkconfig httpd on`).

With this playbook format, Ansible can keep track of the state of everything on all our servers. If you run the playbook the first time, it will provision the server by ensuring Apache is installed and running, and your custom configuration is in place.

Even better, the *second* time you run it (if the server is in the correct state), it won't actually do anything besides tell you nothing has changed. So, with this one short playbook, you're able to provision and ensure the proper configuration for an Apache web server. Additionally, running the playbook with the `--check` option (see the next section below) verifies the configuration matches what's defined in the playbook, without actually running the tasks on the server.

If you ever want to update your configuration, or install another httpd package, either update the configuration file locally or add a package to the `name` list for `yum` and run the playbook again. Whether you have one or a thousand servers, all of their configurations will be updated to match your playbook—and Ansible will tell you if anything ever changes (you're not making ad-hoc changes on individual production servers, *are you*?).

Running Playbooks with `ansible-playbook`

If we run the playbooks in the examples above (which are set to run on `all` hosts), then the playbook would be run against every host defined in your Ansible inventory file (see Chapter 1's basic inventory file example).

Limiting playbooks to particular hosts and groups

You can limit a playbook to specific groups or individual hosts by changing the `hosts:` definition. The value can be set to `all` hosts, a `group` of hosts defined in your inventory, multiple groups of hosts (e.g. `webservers,dbservers`), individual hosts (e.g. `atl.example.com`), or a mixture of hosts. You can even do wild card matches, like `*.example.com`, to match all subdomains of a top-level domain.

You can also limit the hosts on which the playbook is run via the `ansible-playbook` command:

```
$ ansible-playbook playbook.yml --limit webservers
```

In this case (assuming your inventory file contains a `webservers` group), even if the playbook is set to `hosts: all`, or includes hosts in addition to what's defined in the `webservers` group, it will only be run on the hosts defined in `webservers`.

You could also limit the playbook to one particular host:

```
$ ansible-playbook playbook.yml --limit xyz.example.com
```

If you want to see a list of hosts that would be affected by your playbook before you actually run it, use `--list-hosts`:

```
$ ansible-playbook playbook.yml --list-hosts
```

Running this should give output like:

```
playbook: playbook.yml

  play #1 (all): host count=4
    127.0.0.1
    192.168.24.2
    foo.example.com
    bar.example.com
```

(Where `count` is the count of servers defined in your inventory, and following is a list of all the hosts defined in your inventory).

Setting user and sudo options with `ansible-playbook`

If no `remote_user` is defined alongside the `hosts` in a playbook, Ansible assumes you'll connect as the user defined in your inventory file for a particular host, and then will fall back to your local user account name. You can explicitly define a remote user to use for remote plays using the `--user` (`-u`) option:

```
$ ansible-playbook playbook.yml --user=johndoe
```

In some situations, you will need to pass along your sudo password to the remote server to perform commands via `sudo`. In these situations, you'll need use the `--ask-become-pass` (`-K`) option. You can also explicitly force all tasks in a playbook to use sudo with `--become` (`-b`). Finally, you can define the sudo user for tasks run via `sudo` (the default is root) with the `--become-user` (`-U`) option.

For example, the following command will run our example playbook with sudo, performing the tasks as the sudo user `janedoe`, and Ansible will prompt you for the sudo password:

```
$ ansible-playbook playbook.yml --become --become-user=janedoe \
--ask-become-pass
```

If you're not using key-based authentication to connect to your servers (read my warning about the security implications of doing so in Chapter 1), you can use `--ask-pass`.

Other options for `ansible-playbook`

The `ansible-playbook` command also allows for some other common options:

- `--inventory=PATH` (`-i PATH`): Define a custom inventory file (default is the default Ansible inventory file, usually located at `/etc/ansible/hosts`).
- `--verbose` (`-v`): Verbose mode (show all output, including output from successful options). You can pass in `-vvvv` to give every minute detail.
- `--extra-vars=VARS` (`-e VARS`): Define variables to be used in the playbook, in `"key=value,key=value"` format.
- `--forks=NUM` (`-f NUM`): Number for forks (integer). Set this to a number higher than `5` to increase the number of servers on which Ansible will run tasks concurrently.
- `--connection=TYPE` (`-c TYPE`): The type of connection which will be used (this defaults to `ssh`; you might sometimes want to use `local` to run a playbook on your local machine, or on a remote server via cron).

- `--check`: Run the playbook in Check Mode ('Dry Run'); all tasks defined in the playbook will be checked against all hosts, but none will actually be run.

There are some other options and configuration variables that are important to get the most out of `ansible-playbook`, but this should be enough to get you started running the playbooks in this chapter on your own servers or virtual machines.

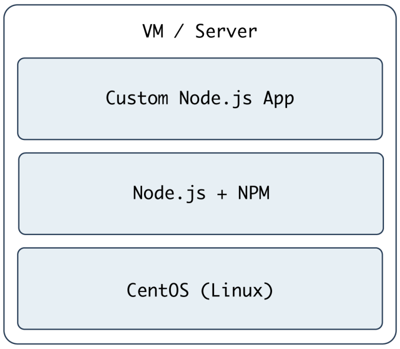
The rest of this chapter uses more realistic Ansible playbooks. All the examples in this chapter are in the Ansible for DevOps GitHub repository[51], and you can clone that repository to your computer (or browse the code online) to follow along more easily. The GitHub repository includes Vagrantfiles with each example, so you can build the servers on your local host using Vagrant.

Real-world playbook: CentOS Node.js app server

The first example, while being helpful for someone who might want to post a simple static web page to a clunky old Apache server, is not a good representation of a real-world scenario. I'm going to run through more complex playbooks that do many different things, most of which are actually being used to manage production infrastructure today.

The first playbook will configure a CentOS server with Node.js, and install and start a simple Node.js application. The server will have a very simple architecture:

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



Node.js app on CentOS.

To start things off, we need to create a YAML file (`playbook.yml` in this example) to contain our playbook. Let's keep things simple:

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

  vars:
    node_apps_location: /usr/local/opt/node

  tasks:
```

First, define a set of hosts (`all`) on which this playbook will be run (see the section above about limiting the playbook to particular groups and hosts), then tell Ansible to run the playbook with root privileges (since we need to install and configure system packages).

Next, we can define `vars` (playbook variables) directly in the playbook; in this case, we are adding the `node_apps_location` variable so we can use that to identify where our Node.js apps will be located.

Finally, the playbook will need to do something on the hosts, so we add a `tasks` section which we'll fill in soon.

Add extra repositories

Adding extra package repositories (yum or apt) is one thing many admins will do before any other work on a server to ensure that certain packages are available, or are at a later version than the ones in the base installation.

In the shell script below, we want to add both the EPEL and Remi repositories, so we can get some packages like Node.js or later versions of other necessary software (these examples presume you're running RHEL/CentOS 7.x as the `root` user):

```
# Install EPEL repo.
yum install -y epel-release

# Import Remi GPG key.
wget https://rpms.remirepo.net/RPM-GPG-KEY-remi \
  -O /etc/pki/rpm-gpg/RPM-GPG-KEY-remi
rpm --import /etc/pki/rpm-gpg/RPM-GPG-KEY-remi

# Install Remi repo.
rpm -Uvh --quiet \
  https://rpms.remirepo.net/enterprise/remi-release-7.rpm

# Install Node.js (npm plus all its dependencies).
yum --enablerepo=epel -y install npm
```

This shell script uses the rpm command to install the EPEL repository, import the Remi repository GPG keys, add the Remi repository, and finally install Node.js. It works okay for a simple deployment (or by hand), but it's silly to run all these commands (some of which could take time or stop your script entirely if your connection is flaky or bad) if the result has already been achieved (namely, two repositories and their GPG keys have been added).

If you wanted to skip a couple steps, you could skip adding the GPG keys, and just run your commands with `--nogpgcheck` (or, in Ansible, set the `disable_gpg_check` parameter of the yum module to `yes`), but it's a good idea to leave this enabled. GPG stands for *GNU Privacy Guard*, and it's a way that developers and package distributors can sign their packages (so you know it's from the original author, and hasn't been modified or corrupted). Unless you *really* know what you're doing, don't disable security settings like GPG key checks.

Ansible makes things a little more robust. Even though the following is slightly more verbose, it performs the same actions in a more structured way, which is simpler to understand, and works with variables and other nifty Ansible features we'll discuss later:

```
    - name: Install EPEL repo.
      yum: name=epel-release state=present

    - name: Import Remi GPG key.
      rpm_key:
        key: "https://rpms.remirepo.net/RPM-GPG-KEY-remi"
        state: present

    - name: Install Remi repo.
      yum:
        name: "https://rpms.remirepo.net/enterprise/remi-release-7.rpm"
        state: present

    - name: Ensure firewalld is stopped (since this is for testing).
      service: name=firewalld state=stopped

    - name: Install Node.js and npm.
      yum: name=npm state=present enablerepo=epel

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

Let's walk through this playbook step-by-step:

1. `yum` installs the EPEL repository (and automatically imports its GPG key).
2. `rpm_key` is a very simple Ansible module that takes and imports an RPM key from a URL or file, or the key id of a key that is already present, and ensures the key is either present or absent (the `state` parameter). We're importing one key, for Remi's repository.
3. We can install extra yum repositories using the `yum` module. Just pass in the URL to the repo `.rpm` file, and Ansible will take care of the rest.
4. Since this server is being used only for test purposes, we disable the system firewall so it won't interfere with testing (using the `service` module).
5. `yum` installs Node.js (along with all the required packages for `npm`, Node's package manager) if it's not present, and allows the EPEL repo to be searched via the `enablerepo` parameter (you could also explicitly *disable* a repository using `disablerepo`).
6. Since NPM is now installed, we use Ansible's `npm` module to install a Node.js utility, `forever`, to launch our app and keep it running. Setting `global` to `yes` tells NPM to install the `forever` node module in `/usr/lib/node_modules/` so it will be available to all users and Node.js apps on the system.

We're beginning to have a nice little Node.js app server set up. Let's set up a little Node.js app that responds to HTTP requests on port 80.

You may be wondering why sometimes quotes are used in these YAML playbooks, and sometimes not. I typically use quotes around my parameters in the following scenarios:

1. If I have a Jinja variable (e.g. `{{ variable_here }}`) at the beginning or end of the line; otherwise YAML will parse the line as nested objects due to the braces.
2. If there are any colons (`:`) in the string (e.g. for URLs).

The easiest way to make sure you're quoting things correctly is to use YAML syntax highlighting in your code editor.

Deploy a Node.js app

The next step is to install a simple Node.js app on our server. First, we'll create a really simple Node.js app by creating a new folder, `app`, in the same folder as your playbook.yml. Create a new file, `app.js`, in this folder, with the following contents:

```
// Load the express module.
var express = require('express');
var app = express();

// Respond to requests for / with 'Hello World'.
app.get('/', function(req, res){
    res.send('Hello World!');
});

// Listen on port 80 (like a true web server).
app.listen(80);
console.log('Express server started successfully.');
```

Don't worry about the syntax or the fact that this is Node.js. We just need a quick example to deploy. This example could've been written in Python, Perl, Java, PHP, or another language, but since Node is a simple language (JavaScript) that runs in a lightweight environment, it's an easy language to use when testing things or prodding your server.

Since this little app is dependent on Express (an http framework for Node), we also need to tell NPM about this dependency via a `package.json` file in the same folder as `app.js`:

```json
{
  "name": "examplenodeapp",
  "description": "Example Express Node.js app.",
  "author": "Jeff Geerling <geerlingguy@mac.com>",
  "dependencies": {
    "express": "4.x"
  },
  "engine": "node >= 0.10.6"
}
```

We need to copy the entire app to the server, and then have NPM download the required dependencies (in this case, `express`), so add these tasks to your playbook:

```yaml
    - name: Ensure Node.js app folder exists.
      file: "path={{ node_apps_location }} state=directory"

    - name: Copy example Node.js app to server.
      copy: "src=app dest={{ node_apps_location }}"

    - name: Install app dependencies defined in package.json.
      npm: path={{ node_apps_location }}/app
```

First, we ensure the directory where our app will be installed exists, using the `file` module.

The `{{ node_apps_location }}` variable used in these tasks was defined under a `vars` section at the top of our playbook, but it could also be overridden in your inventory, or on the command line when calling `ansible-playbook` using the `--extra-vars` option.

Second, we copy the entire app folder up to the server, using Ansible's `copy` command, which intelligently distinguishes between a single file or a directory of files, and recurses through the directory, similar to recursive scp or rsync.

Ansible's `copy` module works very well for single or small groups of files, and recurses through directories automatically. If you are copying hundreds of files, or deeply-nested directory structures, `copy` will get bogged down. In these situations, consider either using the `synchronize` or `rsync` module to copy a full directory, or `unarchive` to copy an archive and have it expanded in place on the server.

Third, we use `npm` again, this time, with no extra arguments besides the path to the app. This tells NPM to parse the package.json file and ensure all the dependencies are present.

We're *almost* finished! The last step is to start the app.

Launch a Node.js app

We'll now use `forever` (which we installed earlier) to start the app.

```
    - name: Check list of running Node.js apps.
      command: forever list
      register: forever_list
      changed_when: false

    - name: Start example Node.js app.
      command: "forever start {{ node_apps_location }}/app/app.js"
      when: "forever_list.stdout.find(node_apps_location + \
'/app/app.js') == -1"
```

In the first play, we're doing two new things:

1. `register` creates a new variable, `forever_list`, to be used in the next play to determine when to run the play. `register` stashes the output (stdout, stderr) of the defined command in the variable name passed to it.
2. `changed_when` tells Ansible explicitly when this play results in a change to the server. In this case, we know the `forever list` command will never change the server, so we just say `false`—the server will never be changed when the command is run.

The second play actually starts the app, using Forever. We could also start the app by calling `node {{ node_apps_location }}/app/app.js`, but we would not be able to control the process easily, and we would also need to use `nohup` and `&` to avoid Ansible hanging on this play.

Forever tracks the Node apps it manages, and we use Forever's `list` option to print a list of running apps. The first time we run this playbook, the list will obviously be empty—but on future runs, if the app is running, we don't want to start another instance of it. To avoid that situation, we tell Ansible when we want to start the app with `when`. Specifically, we tell Ansible to start the app only when the app's path is *not* in the `forever list` output.

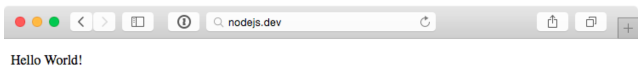
Node.js app server summary

At this point, you have a complete playbook that will install a simple Node.js app which responds to HTTP requests on port 80 with "Hello World!".

To run the playbook on a server (in our case, we could just set up a new VirtualBox VM for testing, either via Vagrant or manually), use the following command (pass in the `node_apps_location` variable via the command):

```
$ ansible-playbook playbook.yml \
--extra-vars="node_apps_location=/usr/local/opt/node"
```

Once the playbook has finished configuring the server and deploying your app, visit `http://hostname/` in a browser (or use `curl` or `wget` to request the site), and you should see the following:



Hello World!

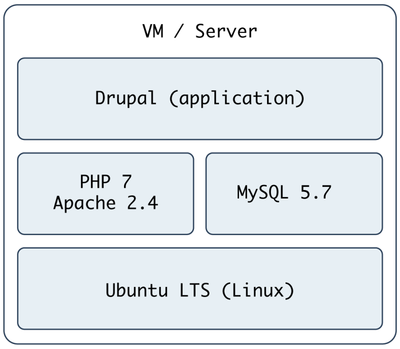
Node.js Application home page.

Simple, but very powerful. We've configured an entire Node.js application server In fewer than fifty lines of YAML!

The entire example Node.js app server playbook is in this book's code repository at https://github.com/geerlingguy/ansible-for-devops[52], in the `nodejs` directory.

Real-world playbook: Ubuntu LAMP server with Drupal

At this point, you should be getting comfortable with Ansible playbooks and the YAML syntax used to define them. Up to this point, most examples have assumed you're working with a CentOS, RHEL, or Fedora server. Ansible plays nicely with other flavors of Linux and BSD-like systems as well. In the following example, we're going to set up a traditional LAMP (Linux, Apache, MySQL, and PHP) server using Ubuntu 16.04 to run a Drupal website.



Drupal LAMP server.

Include a variables file, and discover `pre_tasks` and `handlers`

To make our playbook more efficient and readable, let's begin the playbook (named `playbook.yml`) by instructing Ansible to load in variables from a separate `vars.yml` file:

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

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

  vars_files:
    - vars.yml
```

Using one or more included variable files cleans up your main playbook file, and lets you organize all your configurable variables in one place. At the moment, we don't have any variables to add; we'll define the contents of `vars.yml` later. For now, create the empty file, and continue on to the next section of the playbook, `pre_tasks`:

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

Ansible lets you run tasks before or after the main tasks (defined in `tasks:`) or roles (defined in `roles:`—we'll get to roles later) using `pre_tasks` and `post_tasks`, respectively. In this case, we need to ensure that our apt cache is updated before running the rest of the playbook, so we have the latest package versions on our server. We use Ansible's `apt` module and tell it to update the cache if it's been more than 3600 seconds (1 hour) since the last update.

With that out of the way, we'll add another new section to our playbook, `handlers`:

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

`handlers` are special kinds of tasks you run at the end of a play by adding the `notify` option to any of the tasks in that group. The handler will only be called if one of the tasks notifying the handler makes a change to the server (and doesn't fail), and it will only be notified at the *end* of the play.

To call this handler, add the option `notify: restart apache` after defining the rest of a play. We've defined this handler so we can restart the `apache2` service after a configuration change, which will be explained below.

Just like variables, handlers and tasks may be placed in separate files and included in your playbook to keep things tidy (we'll discuss this in chapter 6). For simplicity's sake, though, the examples in this chapter are shown as in a single playbook file. We'll discuss different playbook organization methods later.

By default, Ansible will stop all playbook execution when a task fails, and won't notify any handlers that may need to be triggered. In some cases, this leads to unintended side effects. If you want to make sure handlers always run after a task uses `notify` to call the handler, even in case of playbook failure, add `--force-handlers` to your `ansible-playbook` command.

Basic LAMP server setup

The first step towards building an application server that depends on the LAMP stack is to build the actual LAMP part of it. This is the simplest process, but still requires a little extra work for our particular server. We want to install Apache, MySQL and PHP, but we'll also need a couple other dependencies.

```
  tasks:
    - name: Get software for apt repository management.
      apt:
        name:
          - python-apt
          - python-pycurl
        state: present

    - name: Add ondrej repository for later versions of PHP.
      apt_repository: repo='ppa:ondrej/php' update_cache=yes

    - name: "Install Apache, MySQL, PHP, and other dependencies."
      apt:
        name:
          - git
          - curl
          - unzip
          - sendmail
          - apache2
          - php7.1-common
          - php7.1-cli
          - php7.1-dev
          - php7.1-gd
          - php7.1-curl
          - php7.1-json
          - php7.1-opcache
          - php7.1-xml
          - php7.1-mbstring
          - php7.1-pdo
          - php7.1-mysql
          - php-apcu
          - libpcre3-dev
          - libapache2-mod-php7.1
          - python-mysqldb
          - mysql-server
        state: present

    - name: Disable the firewall (since this is for local dev only).
      service: name=ufw state=stopped

    - name: "Start Apache, MySQL, and PHP."
      service: "name={{ item }} state=started enabled=yes"
      with_items:
        - apache2
        - mysql
```

In this playbook, we begin with a common LAMP setup:

1. Install a couple helper libraries which allow Python to manage apt more precisely (`python-apt` and `python-pycurl` are required for the `apt_repository` module to do its work).

2. Install an extra `apt` PPA that will allow installation of a later version of PHP than is available in the default system repositories.
3. Install all the required packages for our LAMP server (including all the PHP extensions Drupal requires).
4. Disable the firewall entirely, for testing purposes. If on a production server or any server exposed to the Internet, you should instead have a restrictive firewall only allowing access on ports 22, 80, 443, and other necessary ports.
5. Start up all the required services, and make sure they're enabled to start on system boot.

Configure Apache

The next step is configuring Apache so it will work correctly with Drupal. Out of the box, Apache doesn't have mod_rewrite enabled on Ubuntu's current release. To remedy that situation, you could use the command `sudo a2enmod rewrite`, but Ansible has a handy `apache2_module` module that will do the same thing with idempotence.

We also need to add a VirtualHost entry to give Apache the site's document root and provide other options for the site.

```
    - name: Enable Apache rewrite module (required for Drupal).
      apache2_module: name=rewrite state=present
      notify: restart apache

    - name: Add Apache virtualhost for Drupal 8 development.
      template:
        src: "templates/drupal.test.conf.j2"
        dest: "/etc/apache2/sites-available/{{ domain }}.test.conf"
        owner: root
        group: root
        mode: 0644
      notify: restart apache

    - name: Symlink Drupal virtualhost to sites-enabled.
      file:
        src: "/etc/apache2/sites-available/{{ domain }}.test.conf"
        dest: "/etc/apache2/sites-enabled/{{ domain }}.test.conf"
        state: link
      notify: restart apache

    - name: Remove default virtualhost file.
      file:
        path: "/etc/apache2/sites-enabled/000-default.conf"
        state: absent
      notify: restart apache
```

The first command enables all the required Apache modules by symlinking them from `/etc/apache2/mods-available` to `/etc/apache2/mods-enabled`.

The second command copies a Jinja template we define inside a `templates` folder to Apache's `sites-available` folder, with the correct owner and permissions. Additionally, we `notify` the `restart apache` handler, because copying in a new VirtualHost means Apache needs to be restarted to pick up the change.

Let's look at our Jinja template (denoted by the extra `.j2` on the end of the filename), `drupal.test.conf.j2`:

```
<VirtualHost *:80>
    ServerAdmin webmaster@localhost
    ServerName {{ domain }}.test
    ServerAlias www.{{ domain }}.test
    DocumentRoot {{ drupal_core_path }}
    <Directory "{{ drupal_core_path }}">
        Options FollowSymLinks Indexes
        AllowOverride All
    </Directory>
</VirtualHost>
```

This is a fairly standard Apache VirtualHost definition, but we have a few Jinja template variables mixed in. The syntax for printing a variable in a Jinja template is the same syntax we use in our Ansible playbooks—two brackets around the variable's name (like so: `{{ variable }}`).

There are three variables we will need (`drupal_core_version`, `drupal_core_path`, and `domain`), so add them to the empty `vars.yml` file we created earlier:

```
---
# The core version you want to use (e.g. 8.8.x, 8.9.x).
drupal_core_version: "8.8.x"

# The path where Drupal will be downloaded and installed.
drupal_core_path: "/var/www/drupal-{{ drupal_core_version }}-dev"

# The resulting domain will be [domain].test (with .test appended).
domain: "drupal"
```

Now, when Ansible reaches the play that copies this template into place, the Jinja template will have the variable names replaced with the values `/var/www/drupal-8.8.x-dev` and `drupal` (or whatever values you'd like!).

The last two tasks (lines 75-86) enable the VirtualHost we just added, and remove the default VirtualHost definition, which we no longer need.

At this point, you could start the server, but Apache will likely throw an error since the VirtualHost you've defined doesn't exist (there's no directory at `{{ drupal_core_path }}` yet!). This is why using `notify` is important—instead of adding a task after these three steps to restart Apache (which will fail the first time you run the playbook), notify will wait until after we've finished all the other steps in our main group of tasks (giving us time to finish setting up the server), *then* restart Apache.

Configure PHP with `lineinfile`

We briefly mentioned `lineinfile` earlier in the book, when discussing file management and ad-hoc task execution. Modifying PHP's configuration is a perfect way to demonstrate `lineinfile`'s simplicity and usefulness:

```
    - name: Adjust OpCache memory setting.
      lineinfile:
        dest: "/etc/php/7.1/apache2/conf.d/10-opcache.ini"
        regexp: "^opcache.memory_consumption"
        line: "opcache.memory_consumption = 96"
        state: present
      notify: restart apache
```

Ansible's `lineinfile` module does a simple task: ensures that a particular line of text exists (or doesn't exist) in a file.

In this example, we need to adjust PHP's default `opcache.memory_consumption` option so the Drupal codebase can be compiled into PHP's system memory for much faster page load times.

First, we tell `lineinfile` the location of the file, in the `dest` parameter. Then, we give a regular expression (Python-style) to define what the line looks like (in this case, the line starts with the exact phrase "opcache.memory_consumption"). Next, we tell `lineinfile` exactly how the resulting line should look. Finally, we explicitly state that we want this line to be present (with the `state` parameter).

Ansible will take the regular expression, and see if there's a matching line. If there is, Ansible will make sure the line matches the `line` parameter. If not, Ansible will add the line as defined in the `line` parameter. Ansible will only report a change if it had to add or change the line to match `line`.

Configure MySQL

The next step is to create a database and user (named for the domain we specified earlier) for our Drupal installation to use.

```
    - name: Create a MySQL database for Drupal.
      mysql_db: "db={{ domain }} state=present"

    - name: Create a MySQL user for Drupal.
      mysql_user:
        name: "{{ domain }}"
        password: "1234"
        priv: "{{ domain }}.*:ALL"
        host: localhost
        state: present
```

Ansible works with many databases out of the box (MongoDB, MySQL/-MariaDB, PostgreSQL, Redis and Riak as of this writing). In MySQL's case, Ansible uses the MySQLdb Python package (`python-mysqldb`) to manage a connection to the database server, and assumes the default root account credentials ('root' as the username with no password). Obviously, leaving this default would be a bad idea! On a production server, one of the first steps should be to change the root account password, limit the root account to localhost, and delete any nonessential database users.

If you use different credentials, you can add a .my.cnf file to your remote user's home directory containing the database credentials to allow Ansible to connect to the MySQL database without leaving passwords in your Ansible playbooks or variable files. Otherwise, you can prompt the user running the Ansible playbook for a MySQL username and password. This option, using prompts, will be discussed later in the book.

Install Composer and Drush

Drupal has a command-line companion in the form of Drush. Drush is developed independently of Drupal, and provides a full suite of CLI commands to manage Drupal. Drush, like most modern PHP tools, integrates with external dependencies defined in a `composer.json` file which describes the dependencies to Composer.

We could just download Drupal and perform some setup in the browser by hand at this point, but the goal of this playbook is to have a fully-automated and idempotent

Drupal installation. So, we need to install Composer, then Drush, and use both to install Drupal:

```
    - name: Download Composer installer.
      get_url:
        url: https://getcomposer.org/installer
        dest: /tmp/composer-installer.php
        mode: 0755

    - name: Run Composer installer.
      command: >
        php composer-installer.php
        chdir=/tmp
        creates=/usr/local/bin/composer

    - name: Move Composer into globally-accessible location.
      command: >
        mv /tmp/composer.phar /usr/local/bin/composer
        creates=/usr/local/bin/composer
```

The first two commands download and run Composer's php-based installer, which generates a 'composer.phar' PHP application archive in `/tmp`. This archive is then copied (using the `mv` shell command) to the location `/usr/local/bin/composer` so we can use the `composer` command to install all of Drush's dependencies. The latter two commands are set to run only if the `/usr/local/bin/composer` file doesn't already exist (using the `creates` parameter).

Why use `shell` instead of `command`? Ansible's `command` module is the preferred option for running commands on a host (when an Ansible module won't suffice), and it works in most scenarios. However, `command` doesn't run the command via the remote shell `/bin/sh`, so options like `<`, `>`, `|`, and `&`, and local environment variables like `$HOME` won't work. `shell` allows you to pipe command output to other commands, access the local environment, etc.

There are two other modules which assist in executing shell commands remotely: `script` executes shell scripts (though it's almost always a better idea to convert shell scripts into idempotent Ansible playbooks!), and `raw` executes raw commands via SSH (it should only be used in circumstances where you can't use one of the other options).

It's best to use an Ansible module for every task. If you have to resort to a regular command-line command, try the `command` module first. If you require the options mentioned above, use `shell`. Use of `script` or `raw` should be exceedingly rare, and won't be covered in this book.

Now, we'll install Drush using the latest version from GitHub:

```
    - 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.
      command: >
        /usr/local/bin/composer install
        chdir=/opt/drush
        creates=/opt/drush/vendor/autoload.php

    - name: Create drush bin symlink.
      file:
        src: /opt/drush/drush
        dest: /usr/local/bin/drush
        state: link
```

Earlier in the book, we cloned a git repository using an ad-hoc command. In this case, we're defining a play that uses the `git` module to clone Drush from its repository URL on GitHub. Since we want the `8.x` branch, pass in the `repo` (repository URL), `version` (the branch, tag, or commit ref), and `dest` (destination path) parameters (the `git` module will check out `master` if you don't provide a `version`).

After Drush is downloaded to `/opt/drush`, we use Composer to install all the required dependencies. In this case, we want Ansible to run `composer install` in the directory `/opt/drush` (this is so Composer finds Drush's `composer.json` file automatically), so we pass along the parameter `chdir=/opt/drush`. Once Composer is finished, the file `/opt/drush/vendor/autoload.php` will be created, so we use the `creates` parameter to tell Ansible to skip this step if the file already exists (for idempotency).

Finally, we create a symlink from `/usr/local/bin/drush` to the executable at `/opt/drush/drush`, so we can call the `drush` command anywhere on the system.

Install Drupal with Git and Drush

We'll use `git` again to clone Drupal to the Apache document root we defined earlier in our virtual host configuration, install Drupal's dependencies with Composer, run Drupal's installation via Drush, and fix a couple other file permissions issues so Drupal loads correctly within our VM.

```
    - name: Check out Drupal Core to the Apache docroot.
      git:
        repo: https://git.drupal.org/project/drupal.git
        version: "{{ drupal_core_version }}"
        dest: "{{ drupal_core_path }}"
      register: git_checkout

    - name: Ensure Drupal codebase is owned by www-data.
      file:
        path: "{{ drupal_core_path }}"
        owner: www-data
        group: www-data
        recurse: true
      when: git_checkout.changed | bool

    - name: Install Drupal dependencies with Composer.
      command: >
        /usr/local/bin/composer install
        chdir={{ drupal_core_path }}
        creates={{ drupal_core_path }}/vendor/autoload.php
      become_user: www-data

    - name: Install Drupal.
      command: >
        drush si -y --site-name="{{ drupal_site_name }}"
        --account-name=admin
        --account-pass=admin
        --db-url=mysql://{{ domain }}:1234@localhost/{{ domain }}
        --root={{ drupal_core_path }}
        creates={{ drupal_core_path }}/sites/default/settings.php
      notify: restart apache
      become_user: www-data
```

First, we clone Drupal's git repository, using the `version` defined in our `vars.yml` file as `drupal_core_version`. The `git` module's `version` parameter defines the branch (`master`, `8.2.x`, etc.), tag (`1.0.1`, `8.2.5`, etc.), or individual commit hash (`50a1877`, etc.) to clone.

Before installing Drupal, we must install Drupal's dependencies using Composer (just like with Drush). For both this and the next task, we only want to run them the first time we install Drupal, so we use the `creates` parameter again to let Ansible know if the dependencies have been installed (just like with Drush earlier).

To install Drupal, we use Drush's `si` command (short for `site-install`) to run Drupal's installation (which configures the database (and creates a `sites/default/settings.php` file we can use for idempotence), runs some maintenance, and sets some default configuration settings for the site). We passed in a few variables, like the `drupal_-core_version` and `domain`; we also added a `drupal_site_name`, so add that variable to your `vars.yml` file:

```
# Your Drupal site name.
drupal_site_name: "Drupal Test"
```

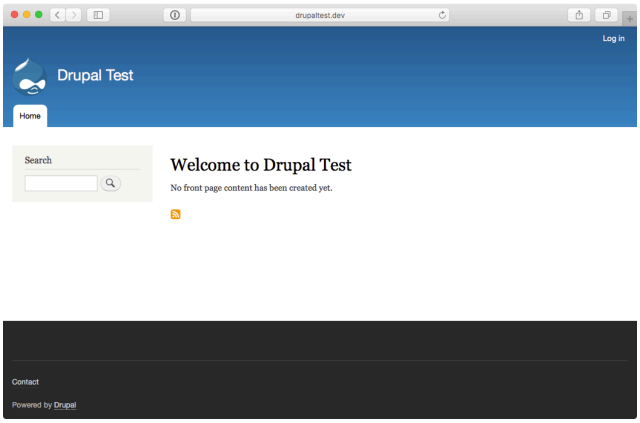
Once the site is installed, we also restart Apache for good measure (using `notify` again, like we did when updating Apache's configuration).

Drupal LAMP server summary

To run the playbook on a server (either via a local VM for testing or on another server), use the following command:

```
$ ansible-playbook playbook.yml
```

After the playbook completes, if you access the server at http://drupal.test/ (assuming you've pointed drupal.test to your server or VM's IP address), you'll see Drupal's default home page, and you could login with 'admin'/'admin'. (Obviously, you'd set a secure password on a production server!).



Welcome to Drupal Test

No front page content has been created yet.

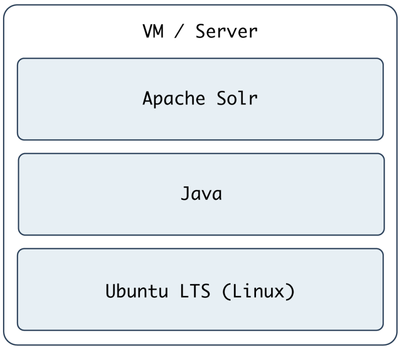
Drupal 8 default home page.

A similar server configuration, running Apache, MySQL, and PHP, can be used to run many popular web frameworks and CMSes besides Drupal, including Symfony, Wordpress, Joomla, Laravel, etc.

The entire example Drupal LAMP server playbook is in this book's code repository at https://github.com/geerlingguy/ansible-for-devops[53], in the `drupal` directory.

Real-world playbook: Ubuntu server with Solr

Apache Solr is a fast and scalable search server optimized for full-text search, word highlighting, faceted search, fast indexing, and more. It's a very popular search server, and it's pretty easy to install and configure using Ansible. In the following example, we're going to set up Apache Solr using Ubuntu 16.04 and Java, on a server or VM with at least 512 MB of RAM.



Apache Solr Server.

Include a variables file, and more `pre_tasks`

Just like the previous LAMP server example, we'll begin this playbook (again named `playbook.yml`) by telling Ansible our variables will be in a separate `vars.yml` file:

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

```
---
- hosts: all
  become: true

  vars_files:
  - vars.yml
```

Let's quickly create the `vars.yml` file, while we're thinking about it. Create the file in the same folder as your Solr playbook, and add the following contents:

```
download_dir: /tmp
solr_dir: /opt/solr
solr_version: 8.2.0
solr_checksum: sha512:beb4e37fc21bf483e3b6bae43cb06a49bc420a0f2b920c979\
09a69a5efeacba1e7d2ff09ae8018446c87bf007f88f06a59de73cd1923f0967e820662\
9b0509b6
```

These variables define two paths we'll use while downloading and installing Apache Solr, and the version and file download checksum for downloading Apache Solr's source.

Back in our playbook, after the `vars_files`, we also need to make sure the apt cache is up to date, using `pre_tasks` like the previous example:

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

Install Java 8

It's easy enough to install Java 8 on Ubuntu, as it's in the default apt repositories. We just need to make sure the right package is installed:

```
  tasks:
    - name: Install Java.
      apt: name=openjdk-8-jdk state=present
```

That was easy enough! We used the `apt` module to install `openjdk-8-jdk`.

Install Apache Solr

Ubuntu's LTS release includes a package for Apache Solr, but it installs an older version, so we'll install the latest version of Solr from source. The first step is downloading the source:

```
    - name: Download Solr.
      get_url:
        url: "https://archive.apache.org/dist/lucene/solr/\
{{ solr_version }}/solr-{{ solr_version }}.tgz"
        dest: "{{ download_dir }}/solr-{{ solr_version }}.tgz"
        checksum: "{{ solr_checksum }}"
```

When downloading files from remote servers, the `get_url` module provides more flexibility and convenience than raw `wget` or `curl` commands.

You have to pass `get_url` a `url` (the source of the file to be downloaded), and a `dest` (the location where the file will be downloaded). If you pass a directory to the `dest` parameter, Ansible will place the file inside, but will always re-download the file on subsequent runs of the playbook (and overwrite the existing download if it has changed). To avoid this extra overhead, we give the full path to the downloaded file.

We also use `checksum`, an optional parameter, for peace of mind; if you are downloading a file or archive that's critical to the functionality and security of your application, it's a good idea to check the file to make sure it is exactly what you're expecting. `checksum` compares a hash of the data in the downloaded file to a hash that you specify (and which is provided alongside the downloads on the Apache Solr website). If the checksum doesn't match the supplied hash, Ansible will fail and discard the freshly-downloaded (and invalid) file.

We need to expand the Solr archive so we can run the installer inside, and we can use the `creates` option to make this operation idempotent:

```
    - name: Expand Solr.
      unarchive:
        src: "{{ download_dir }}/solr-{{ solr_version }}.tgz"
        dest: "{{ download_dir }}"
        remote_src: true
        creates: "{{ download_dir }}/solr-{{ solr_version }}/\
README.txt"
```

If you read the `unarchive` module's documentation, you might notice you could consolidate both the `get_url` and `unarchive` tasks into one task by setting `src` to the file URL. Doing this saves a step in the playbook and is generally preferred, but in Apache Solr's case, the original .tgz archive must be present to complete installation, so we still need both tasks.

Now that the source is present, run the Apache Solr installation script (provided inside the Solr archive's `bin` directory) to complete Solr installation:

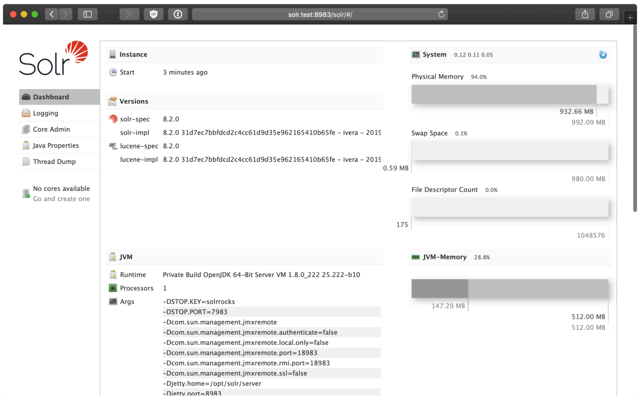
```
    - name: Run Solr installation script.
      command: >
        {{ download_dir }}/solr-{{ solr_version }}/bin/install_solr_ser\
vice.sh
        {{ download_dir }}/solr-{{ solr_version }}.tgz
        -i /opt
        -d /var/solr
        -u solr
        -s solr
        -p 8983
        creates={{ solr_dir }}/bin/solr
```

In this example, the options passed to the installer are hardcoded (e.g. the `-p 8983` tells Apache Solr to run on port `8983`), and this works fine, but if you're going to reuse this playbook for many different types of Solr servers, you should probably configure many of these options with variables defined in `vars.yml`. This exercise is left to the reader.

Finally, we need a task that runs at the end of the playbook to make sure Apache Solr is started, and will start at system boot:

```
    - name: Ensure solr is started and enabled on boot.
      service: name=solr state=started enabled=yes
```

Run the playbook with `$ ansible-playbook playbook.yml`, and after a few minutes (depending on your server's Internet connection speed), you should be able to access the Solr admin interface at http://solr.test:8983/solr (where 'solr.test' is your server's hostname or IP address):



```
Instance
Start          3 minutes ago

Versions
solr-spec      8.2.0
solr-impl      8.2.0 31d7ec7bbfdcd2c4cc61d9d35e962165410b65fe - ivera - 2019
lucene-spec    8.2.0
lucene-impl    8.2.0 31d7ec7bbfdcd2c4cc61d9d35e962165410b65fe - ivera - 2019

JVM
Runtime        Private Build OpenJDK 64-Bit Server VM 1.8.0_222 25.222-b10
Processors     1
Args           -DSTOP.KEY=solrrocks
               -DSTOP.PORT=7983
               -Dcom.sun.management.jmxremote
               -Dcom.sun.management.jmxremote.authenticate=false
               -Dcom.sun.management.jmxremote.local.only=false
               -Dcom.sun.management.jmxremote.port=18983
               -Dcom.sun.management.jmxremote.rmi.port=18983
               -Dcom.sun.management.jmxremote.ssl=false
               -Djetty.home=/opt/solr/server
               -Djetty.port=8983

System    0.12 0.11 0.05
Physical Memory    94.0%
932.66 MB
992.09 MB
Swap Space    0.1%
0.59 MB
980.00 MB
File Descriptor Count    0.0%
175
1048576

JVM-Memory    28.8%
147.29 MB
512.00 MB
512.00 MB
```

Solr Admin page.

Apache Solr server summary

The configuration we used when deploying Apache Solr allows for a multi core setup, so you could add more 'search cores' via the admin interface (as long as the directories and core schema configuration is in place in the filesystem), and have multiple indexes for multiple websites and applications.

A playbook similar to the one above is used as part of the infrastructure for Hosted Apache Solr[54], a service I run which hosts Apache Solr search cores for Drupal websites.

The entire example Apache Solr server playbook is in this book's code repository at https://github.com/geerlingguy/ansible-for-devops[55], in the `solr` directory.

[54]https://hostedapachesolr.com/
[55]https://github.com/geerlingguy/ansible-for-devops

Summary

At this point, you should be getting comfortable with Ansible's *modus operandi*. Playbooks are the heart of Ansible's configuration management and provisioning functionality, and the same modules and similar syntax can be used with ad-hoc commands for deployments and general server management.

Now that you're familiar with playbooks, we'll explore more advanced concepts in building playbooks, like organization of tasks, conditionals, variables, and more. Later, we'll explore the use of playbooks with roles to make them infinitely more flexible and to save time setting up and configuring your infrastructure.

```
 _________________________________________
/ If everything is under control, you are \
\ going too slow. (Mario Andretti)        /
 -----------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```