Automated Server Setup using Ansible

What is Ansible?

Ansible is a platform by Red Hat that has endless automation uses which allows performing actions in multiple servers at the same time. Ansible uses YAML Language for its files and depends on the SSH protocol to manage the machines.

Once Ansible is installed, it will not add a database, and there will be no daemons to start or keep running. You only need to install it on one machine (which could easily be a laptop), and it can manage an entire fleet of remote machines from that central point. When Ansible manages remote machines, it does not leave software installed or running on them, so there's no real question about how to upgrade Ansible when moving to a new version. If you would like to contribute to the project, you can have a look at GitHub

Setting up Ansible

For you to run Ansible, you need a machine running any version of unix, be it Red Hat, Debian, CentOS, macOS, Ubuntu and its flavours and many more. Microsoft Windows is currently not supported. Depending on your local distro, install ansible using your package manager. Ansible is also available via Python pip.

```
## Fedora
$ sudo dnf install ansible

## CentOS
$ sudo yum install ansible

## Ubuntu
$ sudo apt update && sudo apt install -y software-properties-common $ sudo
apt-add-repository --yes --update ppa:ansible/ansible
$ sudo apt install ansible

## local single user installation on macOS or any other system
$ pip install --user ansible

## system wide installation on macOS or any other system
$ sudo pip install ansible
```

We're set! When Ansible is making remote connections, it will use the native SSH service and assumes that you are using SSH keys. It is always advised to use SSH keys over plain password authentication. If you already have a running server but you have not setup passwordless authentication, you can easily copy your public SSH key using `ssh-copy-id` on your local machine. Substitute the server IP address (and the user if you want passwordless authenticaion for a different user) in the command below.

```
$ ssh-copy-id root@123.45.6.78
```

Now we've installed Ansible. We need a fresh server with nothing installed so that we can automate the manual process we completed in the previous section. Once we spin up a new droplet from Digital Ocean (or any other provider, or a local server), we can start playing around with Ansible. Make an

ansible directory somewhere awesome on your local machine. In our case, we will use `~/devops`.

Ansible Inventory

Ansible allows us to list our server IP addresses in different formats. Let's use both INI and YAML formats to explain the difference. Inside the new directory, create two new files; `hosts` and `hosts.yml`; to keep an inventory of all our server IP addresses. Since we have not setup another user for our servers, we shall use the root user. We can also give each of our servers names and group them as per our needs.

```
$ cat hosts

# your server ip addresses
1.x.x.x
2.x.x.x
[databases]
3.y.y.y
4.y.y.y
[php]
server ansible_host=5.z.z.z
worker ansible_host=6.z.z.z ansible_port=8888
```

In the INI file above the:

- `x.x.x` addresses are only in the `all` and `ungrouped` groups,
- `y.y.y` in the `databases` group and,
- `z.z.z` in the `php` group.

The two PHP server groups have aliases (server and worker) which can be used when you explicitly want to target either of the two servers. The worker server also has a custom SSH port `8888` which must be defined using the `ansible_port` variable.

The YAML equivalent of the INI above is:

```
$ cat hosts.yml

# your server ip addresses
all:
  hosts:
    1.x.x.x:
    2.x.x.x:
  children:
    databases:
      hosts:
        3.y.y.y:
        4.y.y.y:
    php:
      hosts:
        server:
          ansible_host: 5.z.z.z
        worker:
          ansible_host: 6.z.z.z
          ansible_port: 8888
```

For the Ansible hosts, the colon at the end of each list item is required when the hosts are more than one thus, it's always advised to keep the colon with or without multiple hosts. To test our inventory, ping any of the listed servers using the ansible ping module:

```
$ ansible all -i hosts -m ping
$ ansible all -i hosts.yml -m ping
$ ansible databases -i hosts -m ping
$ ansible databases -i hosts.yml -m ping
$ ansible worker -i hosts -m ping
$ ansible worker -i hosts.yml -m ping
...
6.z.z.z | SUCCESS => {
  "ansible_facts": {
    "discovered_interpreter_python": "/usr/bin/python3"
  },
  "changed": false,
  "ping": "pong"
}
```

YAML Basics

The YAML syntax is pretty straight forward. We'll go through a few basics before having a look at playbooks.

As a common convention, Ansible files can begin with `---` and end with `...` denoting the start and end of the document. Most files will only include the beginning marker. Comments are defined by the `#` character.

Lists are very common in Ansible and are defined by using `-` (hyphen) followed by a space.

```
---
- php
- vue
- react
```

Dictionaries are `key: value` pairs in which the colon must always be followed by a space.

```
php:
    latest: 7.3
    Installed: 7.3
```

In the case that you have a line that is too long, it can be wrapped using the **Literal Block Scalar character** | (pipe character), or the **Folded Block Scalar character** > (greater than character). The Literal Block Scalar character will render the input as typed, literally including the new lines and any additional spaces while the Folded Block Scalar character will convert the new lines into spaces. New lines can be enforced by either indenting the line or adding an extra empty line.

```yaml
# Literal
exact_hello_world_template: |
                        this is
                        the
                        hello world template
# same as:
# this is\nthe\nwrapped\nhello world template
```

```yaml
# Folded
wrapped_hello_world_template: >
                        this is
                        the
                        wrapped
                        hello world template
# same as:
# this is the wrapped\nhello world template
```

Notice: In case your strings contain a colon or a #, you will need to quote them as YAML will expect a mapping when a colon is followed by a space i.e. John Doe: Look for him. Anything after the # will be regarded as a comment.

Ansible Playbooks

Playbooks are Ansible's configuration, deployment, and orchestration language. They can describe a policy you want your remote systems to enforce, or a set of steps in a general IT process and in our case, how we want our servers to be setup.

Each playbook consists of tasks, which are a set of steps that should be performed on the hosts that you select. The tasks are executed in the order that they are written against the selected hosts, one task at a time. In order to make our playbooks configurable, we must use variables within our tasks. These variables can be defined in the Ansible Inventory file or the play or even in the task.

When the tasks are executed, the name of the task is displayed as part of the output. Each task within the play should therefore have a name, a simple human readable description of the task.

We can group our tasks by following any structure we prefer. Since we are setting up a LEMP server, we can have `php`, `mysql`, `nginx`, `common` and `acl` groups. In ansible these groups are referred to as `roles`. The good thing with roles is that they are self-contained thus, easily sharable between playbooks.

We're good to go! In the `~/devops` directory, add a new `site.yml` file. This will be our play definition file. Next we need a directory to store all our roles; create the `roles` directory. We will then use `ansible-galaxy` to create our 5 roles.

```
$ touch site.yml
$ mkdir roles
$ cd roles
$ ansible-galaxy init php
$ ansible-galaxy init mysql
$ ansible-galaxy init nginx
$ ansible-galaxy init common
$ ansible-galaxy init acl
```

First, let's give Ansible the list of hosts to use. Define your Ansible Inventory in the format you prefer. I prefer using the INI format:

```
123.45.6.78 ansible_user=root
```

> **Notice:** Since we are using the `root` account in our server, we will have to explicitly instruct ansible by adding the `ansible_user=root` parameter; failure to which, it will try to connect using the local user account and fail.

Next, open `site.yml` using your favourite editor. I prefer Visual Studio Code with the YAML and Ansible extensions.

> **Notice:** Each playbook consists of a list of plays and each play executed against a list or servers (hosts).

The first directive we shall give is a list of hosts that the play should be executed on; `all` should do for now.

```
---
- hosts: all
```

Next we'll define a section to hold variables that will be used by the play. This section is defined by the key `vars` and should have its properties indented from the margin since it's a dictionary.

```
---
- hosts: all
  vars:
    KEY: sample value
```

We can now test whether the playbook is set up correctly by adding a ping task that will do the same thing as running `ansible all -i hosts -m ping`.

```
---
- hosts: all
  vars:
    KEY: sample value
  tasks:
    - name: Ping our awesome server
      ping:
        data: pong
```

Open up the terminal and navigate to the `~/devops` directory and execute the playbook

```
$ cd ~/devops
$ ansible-playbook -i ./hosts site.yml
PLAY [all] *********************************************

TASK [Gathering Facts] *********************************
ok: [123.45.6.78]

TASK [Ping our awesome server] *********************
ok: [123.45.6.78]

PLAY RECAP *********************************************
123.45.6.78 : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
```

Congratulations! You've run your first playbook. Looking back, the playbook has a similar syntax to the YAML hosts file. We will be using YAML for all our playbook information including the roles.

Remove the tasks that we just added and replace them with our roles. Depending on your setup, you will need to have more than one play in the playbook, each with its own set of hosts. In our case, we are using one common server for all our services.

When running the play, we will execute the `common` role first so that we do the basic server setup and perform system upgrades. We plan on using PHP-FPM via a unix socket. The path of the unix socket is dependent on the PHP version we are running. If we choose `PHP 7.3`, the socket will be `unix:/run/php/php7.3-fpm.sock` while `PHP 7.4` will give us

`unix:/run/php/php7.4-fpm.sock`. The socket location is used by both Nginx and PHP. The PHP version also determines the PHP modules to be installed. If we are using `PHP 7.3`, we will have to install the `php7.3-*` modules. In light of all this, it makes perfect sense to keep the PHP version as a variable. So let's add it as our first `var`.

```
---
- name: Spinning up our LEMP server.
  hosts: all
  vars:
    php_version: "7.3"
  roles:
    - common
    - mysql
    - php
    - nginx
    - acl
```

When using different hosts you can use:

```
---
- name: Common Setup
  hosts: all
  roles:
    - common
    - security
    - ...
    - ...
- name: Spin up php
  hosts:
    - php
  vars:
    php_version: "7.3"
  roles:
    - php
```

Great! Let's get started on our roles and check out their structure.

Role Structure

Each role that is created using `ansible-galaxy init role` comes with a set of folders. Not all the folders are required but at the least, the `tasks` folder should be present. Below are the default folders that you get:

- Defaults => stores the default variables that will be used in the role in case a variable is undefined.
- Files => keeps the deployable files that are to be used in the role.
- Handlers => keeps the handlers (event listeners) that can be used within or without the role.
- Meta => defines some metadata that will be used by the role.
- Tasks => keeps the list of tasks to be executed by the role.
- Templates => contains templates that can be edited and deployed by the role.
- Vars => contains variables that are used by the role.

You can raise events in Ansible which will be handled by the listeners that are defined in the `handlers` directory. All our tasks will be defined and stored in the `tasks` directory.

Laravel Envoy

Laravel Envoy will be used to configure our server and generate any new directories and releases. It provides a clean, minimal syntax for defining common tasks you run on your remote servers. Using Blade style syntax, you can easily setup tasks for deployment, Artisan commands, and more.

We shall instruct it to create the app directories if they are missing and if the `persistent` directory is missing, it should first clone the repository to the `releases` directory, then copy over the needed files and directories to the `persistent` directory. Once the deployment is done, the script should remove the current directory and symlink it to the latest release.

To get started, create a new `Envoy.blade.php` file at the root of the devops Laravel app directory. In this file, we shall define the set of servers we are going to deploy to with their corresponding SSH ports. To make the envoy file configurable and reusable, we shall execute it with two options; the first will be the **GitLab repository**, and the second will be a **JSON encoded list of servers**. These two options will allow the envoy file to be used across repositories without the need of changing the file.

Before executing anything we shall do some setup. In the `setup` section, we'll need to validate that the two options have been set. Secondly, we will setup the `base paths` as per our directory structure and check if the `persistent` storage exits so that we can create it if missing. After the setup, we shall define our `servers list` using the json decoded list of servers.

```
@setup
    if (!$repo || !$servers) {
        echo 'The repo and servers parameter is required';
        die();
    }

    $servers = json_decode($servers, true, JSON_THROW_ON_ERROR);
    if (!$servers || count($servers) < 1) {
        echo 'servers parameter should be a valid json array of servers.';
        die();
    }

    $persistentDir = '/var/www/app/persistent';
    $releasesDir = '/var/www/app/releases';
    $appDir = '/var/www/app/current';
    $release = date('YmdHis');
    $newReleaseDir = $releasesDir .'/'. $release;
    $persistentStorageExists = is_dir($persistentDir . '/storage');
@endsetup

@servers($servers)
```

We can now setup our `deploy story` that will be called by the GitLab runner. You can have as many stories as needed. In our case, we only need the `deploy story` which will handle:

- the cloning of the new release,
- linking of the persistent storage and env file,
- installing of composer (and yarn/npm) dependencies,
- linking of the public storage directory,
- migrating the database,
- caching routes and configs,
- activating the release as the current one and,
- finally removing old releases.

```
@story('deploy')
    cloneRelease
    linkPersistentStorage
    installDependencies
    optimiseRelease
    activateRelease
    cleanup
@endstory
```

Each item in the story above is a task that will be run on each of the servers defined in the `servers array`. The good thing about Envoy is that you can incorporate your variables easily in any command you are running. Let's setup all the tasks, performing the steps as we would manually do it. I have commented out the horizon termination since we have not setup supervisor and horizon. Checkout the docs. The complete Envoy.blade.php file:

```
@setup
if (!$repo || !$servers) {
exit('The repo and servers parameter is required');
}
$servers = json_decode($servers, true, JSON_THROW_ON_ERROR);
if (!$servers || count($servers) < 1) {
exit('servers parameter should be a valid json array of servers.');
}
$persistentDir = '/var/www/app/persistent';
$releasesDir = '/var/www/app/releases';
$appDir = '/var/www/app/current';
$release = date('YmdHis');
$newReleaseDir = $releasesDir .'/'. $release;
$persistentStorageExists = is_dir($persistentDir . '/storage');
@endsetup

@servers($servers)

@story('deploy')
cloneRelease
linkPersistentStorage
installDependencies
optimiseRelease
activateRelease
@endstory
```