Learning Objectives

Learners will be able to...

- Deploy WordPress on our local server instance using Ansible
- Connect the instance to a database for WordPress to store its data using Ansible
- Compose and upload your own Ansible project to GitHub

|||info
Make Sure You Know
All Ansible concepts covered in the course

Connecting to GitHub

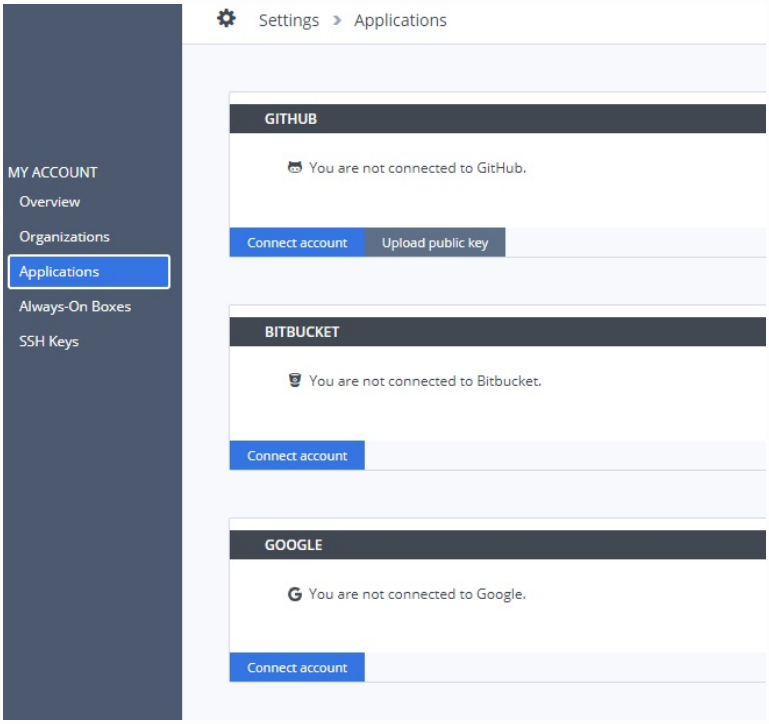
Connecting to GitHub

In this lab, you are going to deploy WordPress on a local server instance using Ansible. In order to showcase this project for your portfolio, you are going to use GitHub. If you do not yet have an account, please create one now. We are going to clone a repo that will contain the code for your Ansible configuration.

Connecting GitHub and Codio

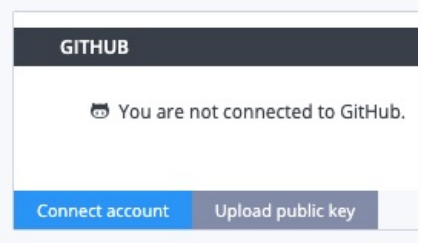
You need to connect GitHub to your Codio account. This only needs to be done one time. Follow the connection guide, or follow these steps:

- In your Codio account, click on your username
- Click on **Applications**



Codio Account

- Under GitHub, click on **Connect account**



connect account

- You will be using an SSH connection, so you need to click on **Upload public key**

Fork the Repo

- Go to the Ansible-Capstone repo. This repo is the starting point for your project.
- Click on the "Fork" button in the top-right corner.
- Click the green "Code" button.
- Copy the **SSH** information. It should look something like this:

```
git@github.com:<your_github_username>/Ansible-Capstone.git
```

info

Important

If you do not use the SSH information, you will have to provide your username and Personal Access Token (PAT) to GitHub each time you push or pull from the repository. See this documentation for setting up a PAT for your GitHub account.

In the Terminal

- Clone the repo. Your command should look something like this:

```
git clone git@github.com:<your_github_username>/Ansible-
        Capstone.git
```

- You should see a `Ansible-Capstone` directory appear in the file tree.

On the last page, we'll show you how to push to Github.

You are now ready for the lab!

Step 1: Project Intro and Local Variables

Project Description

Our goal is to configure, deploy, and install WordPress on a local instance using Ansible.

> info
>
> **What is WordPress ?**
>
> WordPress is a very popular content management system (CMS). It is claimed to be used for over 30% of all websites.

WordPress Requirements

WordPress requires an SQL database to store its data. To satisfy this requirement, we will write our Ansible configuration to provision a local database.

This demo will guide you, step by step, through creating your own portfolio-worthy project to showcase your knowledge of Ansible!

Let's begin!

Step 1: Define Local Variables

Let's start by defining some local variables that we will be using in our playbooks.

You should see a blank `vars/main.yml` file open in the left panel; the variables will be propagated to roles, overriding default values in `default/main.yml` automatically when the role is included. Paste the following code in that file:

```
wordpress_db: wordpress
wordpress_user: wordpress
wordpress_password: wordpress_password
mysql_root_password: super-secure-password

mysql_databases:
  - name: "{{wordpress_db}}"
mysql_users:
  - name: "{{wordpress_user}}"
    host: "%"
    password: "{{wordpress_password}}"
    priv: "{{wordpress_user}}.*:ALL"
```

We've defined a couple variables for the database that our WordPress instance will need, including those for the role, database, and user:
- `wordpress_db` and `wordpress_user` specify the name of the MySQL database and user, respectively, that will be created for WordPress
- `wordpress_password` specifies the password for the MySQL user
- the variables `mysql_databases` and `mysql_users` are used by the my_sql role

> info
>
> You may have noticed that `wordpress_db` and `wordpress_user` are given the same name, `wordpress`. This is not common practice and is done to simplify this project.
>
> It's typical practice for the database to be called the same as the website, with the user having a separate name, like `admin`.
>
> It's due to our project not deploying to a website but rather a local host that this is our preferred nomenclature.

Let's move on to installing dependencies!

Step 2: Dependencies

Installing Dependencies:

WordPress requires a database, so we will install MySQL and configure it using the third party role `geerlingguy.mysql` (the Ansible Galaxy link is here if you want to learn more).

info

Why does WordPress need a database?

WordPress is a content management system (CMS) that is designed to dynamically generate web pages by fetching data from a database, making a relational (SQL) database is an essential part of any WordPress site. The database stores account information, all the website content, as well as it's configuration settings.

When a user requests a web page, WordPress fetches the relevant data from the database and uses it to generate the requested page, making the website functional, smooth, and optimized.

Our playbook has its own dependencies, which we need to specify in the `requirements.yml` file, currently empty on shown on the left. We want to add the `geerlingguy.mysql` role from Ansible Galaxy, so copy the following code block into the requirements file:

```
roles:
  - name: geerlingguy.mysql
```

This role will install the MySQL dependencies locally, but by default it will not be installed to a local folder so you won't see it appear in the file tree. To install, run the command below in the terminal:

```
ansible-galaxy install -r Ansible-Capstone/requirements.yml
```

This command will install the dependencies (in this case the `geerlingguy.mysql` role) into our local box.

With dependencies installed, we are ready to write our own custom role to install WordPress; Step 3 will guide you through the role assembly.

Step 3a: WordPress Role

WordPress Requirements

Step 3a covers the installation and configuration of the Apache web server and the required PHP packages, which are required to run WordPress.

Installing Required Packages for WordPress (Lines 1-27)

There are a few required dependencies for running WordPress (found easily by web search). Our first role task should be installing those packages using the `apt` module:

```
- name: Install Packages for wordpress
  ansible.builtin.apt:
    name:
      - apache2
      - php
      - php-pear
      - php-cgi
      - php-common
      - php-curl
      - php-mbstring
      - php-gd
      - php-mysqlnd
      - php-bcmath
      - php-json
      - php-xml
      - php-intl
      - php-zip
      - php-imap
      - php-imagick
    update_cache: yes
```

> important
>
> `update_cache: yes` updates the package index for the repository so that the installation does not fail if the old package was removed from the repository.

Installation and Setup of Apache (Lines 22-27)

We need the Apache2 service to be enabled and start automatically with the system, so we will use the `systemd` module to do so.

```
- name: Enable service httpd and ensure it is not masked
  ansible.builtin.systemd:
    name: apache2
    enabled: true
    masked: no
    state: started
```

▼ **Lines 22-27 Breakdown:**

- `name: Enable service httpd and ensure it is not masked` is the name of the task and identifies its purpose
- `ansible.builtin.systemd:` is the Ansible module used to manage the system and service manager in Linux.
- `name: apache2`: This is the name of the service to be enabled, in this case the Apache2 service.
- `enabled: true`: This parameter ensures that the service is enabled, meaning it will start automatically at boot time.
- `masked: no`: This parameter ensures that the service is not masked. A masked service cannot be started, which is useful for preventing accidental service starts.
- `state: started`: This parameter ensures that the service is started immediately.

Once this task is executed, Apache2 will start automatically on boot time, and it will be in the “started” state, meaning that it is currently running.

Ownership, Group, and Permissions(Lines 29-44)

The next task we include in the role changes the file ownership, group, and permissions of the `/var/www/html` directory and its contents. We use the `file` module to do so:

```yaml
- name: Change file ownership, group and permissions
  ansible.builtin.file:
    path: /var/www/html
    owner: codio
    group: www-data
    mode: '0664'
    recurse: true
```

▼ **Lines 29-35 Breakdown:**

- `name: Change file ownership, group and permissions` is the name of the task for identification purposes
- `ansible.builtin.file` is the Ansible module used to manage files and directories
- `path: /var/www/html` specifies the path of the directory that we want to modify
- `owner: codio` sets the owner of the directory to the user codio; in this case, the `codio` user is the owner because the role is being executed by a user with that name
- `group: www-data` sets the group of the directory to `www-data`
- `mode: '0664'` sets the permissions of the directory to `0664`, which means that the owner and group have read and write permissions, and everyone else has only read permissions. Note: the single quotes around the mode value indicate that it should be treated as a string
- `recurse: true` specifies that the file module should modify the ownership and permissions of all files and directories within the specified path recursively

Step 3b: WordPress Role

Installing and Configuring WordPress

The tasks you will include in the WordPress role in Step 3b will install and configure the WordPress application

Unarchiving WordPress (Lines 37-44)

Our next task downloads and extracts the WordPress archive file from the specified source URL, places the extracted files in the `/var/www/html/` directory on the target system, and sets the ownership and group of the extracted files to `codio` and `www-data`, respectively. To do this we use Ansible's `unarchive` module:

```yaml
- name: Unarchive wordpress
  ansible.builtin.unarchive:
    src: https://wordpress.org/latest.tar.gz
    dest: /var/www/html/
    remote_src: yes
    owner: codio
    group: www-data
    extra_opts: [--strip-components=1]
```

▼ **Lines 37-44 Breakdown:**

- `name: Unarchive wordpress` is the name of the task
- `ansible.builtin.unarchive` is the Ansible module used to unarchive files.
- `src: https://wordpress.org/latest.tar.gz` specifies the source URL of the WordPress archive file to be downloaded and unarchived.
- `dest: /var/www/html/` specifies the destination directory where the archive file will be extracted.
- `remote_src: yes` specifies that the archive file should be downloaded from the remote source (i.e., the URL specified in src) to the target system before unarchiving.
- `owner: codio` This sets the owner of the extracted files to the user codio.
- `group: www-data` sets the group of the extracted files to `www-data`.
- `extra_opts: [--strip-components=1]` specifies any extra options to pass to the `tar` command during extraction
  - In this case, the `--strip-components=1` option is used to remove the first component of the file path when extracting the archive which

is useful for ensuring that the extracted files are placed directly in the destination directory (`/var/www/html/`) rather than in a subdirectory within that directory.

Updating Config File (Lines 46-63)

As discussed in the previous **File Management** assignment, we will be using the `template` module to generate the actual configuration file during the deployment process. This file, `wp-config.php.j2`, has been provided for you and is located in the `/roles/wordpress/templates/` directory.

```
- name: Update Config
  template:
    src: wp-config.php.j2
    dest: /var/www/html/wp-config.php
```

▼ **Lines 46-49 and Template File Breakdown:**

To implement the `template` module we specify the source template file (`wp-config.php.j2`), and destination path of the generated configuration file (`/var/www/html/wp-config.php`).

The template file for the WordPress configuration file, `wp-config.php`, contains the default configuration settings for WordPress, including database settings, secret keys, database table prefix, and the absolute path to the WordPress directory.

In this specific template file, there are placeholders like `{{wordpress_db}}`, `{{wordpress_user}}`, and `{{wordpress_password}}`, which are replaced with actual values during the deployment process. These values are specific to the environment where the WordPress application is being deployed, which we defined in the `vars/main.yml` file.

Updating Ownership and Permissions (Lines 51-58)

We are downloading directly from WordPress, so the files and directories will have their own ownership and permissions. We've already set file permissions, so our next task will be to change the ownership and permissions of the `var/www/html` directory, the root directory for the WordPress installation:

```
- name: Change file ownership, group and permissions
  ansible.builtin.file:
    path: /var/www/html
    owner: codio
    group: www-data
    mode: u=rwX,g=rwX,o=rX
    recurse: true
    state: directory
```

▼ **Lines 51-58 Breakdown:**

The `owner: codio` and `group: www-data` parameters are setting the user and group ownership of the directory to `codio` and `www-data`, respectively. This means that the `codio` user will have full read, write, and execute permissions to the directory, and members of the `www-data` group will have read and write permissions.

The `mode: u=rwX,g=rwX,o=rX` parameter is setting the permissions for the directory to `u=rwX,g=rwX,o=rX`. This means that the user, group, and others will have the following permissions:
- `r`: read permission
- `w`: write permission
- `X`: execute permission

So the owner (`codio`) and group (`www-data`) will have read, write, and execute permission, while others will only have read and execute permission.

The `recurse: true parameter` is telling Ansible to apply these ownership and permission changes recursively to all files and subdirectories within the `/var/www/html` directory.

The `state: directory` parameter is specifying that these changes should only be applied to the `/var/www/html` directory if it is a directory (as opposed to a file or symlink).

Removing Index File (Lines 60-63)

By default, when Apache is installed, it creates an index file located in the `/var/www/html` directory. The following task will remove this file to avoid conflicts:

```
- name: remove old index
  ansible.builtin.file:
    path: /var/www/html/index.html
    state: absent
```

▼ **Lines 60-63 Breakdown:**

The `ansible.builtin.file` module takes the path to the file to be operated on as the `path` parameter, and the desired state of the file as the `state` parameter. In this case, the `state` parameter is set to `"absent"`, which means that the file should be removed if it exists.

This task is intended to ensure that the default `index.html` file, which may have been created during the installation of the Apache web server, is removed from the `/var/www/html` directory.

Step 3c: WordPress Role

Configuring Apache

The remaining three tasks you will add the WordPress role will configure Apache for use with our specific use case.

Enabling .htaccess Files (Lines 65-76)

This task enables the use of `.htaccess` files in the Apache configuration by replacing the “AllowOverride None” directive with “AllowOverride All”, and also sets two RequestHeader directives for X-Forwarded-Proto and X-Forwarded-Port. The `notify` keyword triggers the `restart apache2` handler if the task makes any changes.

```
- name: Enable .htaccess files in Apache config
  ansible.builtin.replace:
    path:  /etc/apache2/apache2.conf
    after: '<Directory /var/www/>'
    before: '</Directory>'
    regexp: 'AllowOverride None'
    replace: |
      AllowOverride All
      RequestHeader set X-Forwarded-Proto "https"
      RequestHeader set X-Forwarded-Port "443"
  notify:
    - restart apache2
```

▼ **Lines 65-76 Breakdown:**

This task enables the use of .htaccess files in the Apache web server configuration file (`/etc/apache2/apache2.conf`) by replacing the line “AllowOverride None” with “AllowOverride All”.

It also adds two RequestHeader lines that sets the values for the X-Forwarded-Proto and X-Forwarded-Port HTTP headers. These headers are used when Apache is run behind a reverse proxy, and can be used to determine the protocol and port used by the client.

Finally, it includes a notification to restart the Apache service to apply the changes.

Enabling Apache `mod_headers` (Lines 78-84)

Our next Ansible task creates a symbolic link (symlink) to enable the Apache module “mod_headers” using the `file` module:

```
- name: Create a symbolic link for mod_headers
  ansible.builtin.file:
    src: /etc/apache2/mods-available/headers.load
    dest: /etc/apache2/mods-enabled/headers.load
    state: link
  notify:
    - restart apache2
```

▼ **Lines 78-84 Breakdown:**

The task uses the `ansible.builtin.file` module to create the link by specifying the source file (`/etc/apache2/mods-available/headers.load`) and the destination file (`/etc/apache2/mods-enabled/headers.load`). The `state: link` parameter specifies that the destination file should be a symbolic link to the source file.

lastly, the `notify` parameter specifies that the task should send a notification to the `restart apache2` handler to restart the Apache service if the task changes the state of the system.

Replacing Default Port (Lines 86-92)

The default port for a Codio preview is port `3000`, so we will use the `replace` module to replace Apache’s default port `80`:

```
- name: Enables port 3000
  ansible.builtin.replace:
    path:  /etc/apache2/ports.conf
    regexp: 'Listen 80'
    replace: Listen 3000
  notify:
    - restart apache2
```

▼ **Lines 89-92 Breakdown:**

The `ansible.builtin.replace` module replaces the default port `80` used by the Apache web server with `3000`. The file that is edited is the `ports.conf` file, located at `/etc/apache2/ports.conf`. The `regexp` parameter is used to find the line in the file that specifies `Listen 80`, and the `replace` parameter

is used to replace it with `Listen 3000`. Finally, the `notify` parameter triggers the `restart apache2` handler to restart the Apache web server to apply the changes made.

You may have noticed the `notify` parameter included in the final three tasks to call our `restart apache2` handler; these tasks change configuration files or enable a module, so they require a restart of Apache.

Next, we'll write the `restart apache2` handler.

Step 4: Handlers

The `restart apache2` Handler

The tasks that change configuration files or enable a module require a restart of Apache, so we need to include a handler in the `wordpress` role to accomplish this.

On the left you'll see the blank handlers file `roles/wordpress/handlers/main.yml`. Add the following code to it:

```
- name: restart apache2
  ansible.builtin.systemd:
    name: apache2
    state: restarted
```

We are installing our MySQL and Apache2 services on a local server that runs Ubuntu, which is a Linux-based operating system, so Systemd will be our default service manager. We will need to use the `ansible.builtin.systemd` module to restart these services, so this is the only handler that we need to include

> info
>
> There are other handlers for MySQL, but they are inside the `geerlingguy.mysql` role.

Step 5: WordPress Playbook

Assembling our WordPress Playbook

In addition to using the `geerlingguy.mysql` roles, we'll include our custom WordPress role to install WordPress and MySQL on our local Codio box, `localhost`.

Let's go through each section of the playbook:

Hosts, Privileges, and Variables:

First we'll specify the local host, the requisite privileges, and the variables we want to include.

```
- hosts: localhost
  become: yes
  vars_files:
    - vars/main.yml
```

- `hosts: localhost` specifies that the playbook will be run on the localhost machine
- `become: yes` specifies that Ansible should run the tasks in the playbook with elevated privileges
- `vars_files` is a list of YAML files that contain variables to be used in the playbook. In this case, our `vars/main.yml` file will be loaded so the variables we defined in that file will be available for use in the playbook

Roles

We are implementing two roles: `wordpress` to install apache and PHP, and `geerlingguy.mysql` to install MySQL.

```
  roles:
    - role: wordpress
    - role: geerlingguy.mysql
```

- `roles: - { role: wordpress } - { role: geerlingguy.mysql }` specifies the roles to be executed in the playbook. The roles will be executed in that order, though in our use case the order is unimportant

they are each setting up independent things.

Run and Reset

Run the Playbook

Now that we have our configuration finalized, it's time to apply it and deploy our `instance` and `database`.

Let's do that by running our `wordpress.yml` playbook:

```
ansible-playbook Ansible-Capstone/wordpress.yml
```

Resetting the Box

If something goes wrong during your deployment, you can always restart your project box to try it again. However, resetting a box is fairly destructive and should be used with caution.

When performing a box reset, the box is returned to a fresh state but your code files are untouched. It also results in the following:

- Any new folders or files that have been created will be deleted.
- All parts of the box outside the ~/workspace folder will be reset.
- All code files in the ~/workspace folder are untouched.

To restart the box, click the Project tab and choose Restart Box. The reboot normally takes a few seconds.

Lastly, let's push our project to GitHub!

Pushing to GitHub

Pushing to GitHub

Before continuing, you must push your work to GitHub.
In the terminal:

- Commit your changes:

```
git add .
git commit -m "Completed the config to deploy WordPress"
```

- Push to GitHub:

```
git push
```