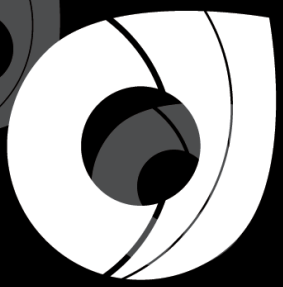
Corndel DevOps Engineering Programme

in association with Softwire

Module 4: Project Exercise



Corndel Digital.

DA9I

Module 4

Project Exercise Brief

In this exercise you'll configure your To Do app to run in a virtual machine using Vagrant. Vagrant encapsulates your development environment in a single configuration file, making it easy to share between developers and launch with a single command - `vagrant up` - without having to worry about Python environments and dependencies.

This exercise will walk through the key steps and requirements in this process. Setting up vagrant can be tricky, and there are always pitfalls in installing a full development environment on a new machine, but it can save a lot of effort in the long run.

Prerequisites

Hypervisor

Vagrant requires a hypervisor installed. We recommend VirtualBox.

Vagrant

Download and install vagrant from the official website. You can check it's installed correctly by running the `vagrant` command in your terminal.

Hints

If you find yourself stuck, there are some hints and suggestions at the end of this document. You can also research the topic online.

This section will guide you through setting up Vagrant to run your To Do app. If you're already feeling confident with managing Python and Linux environment setup, we recommend you give it a try yourself before reading through the material!

Submitting your work

We'll be using Pull Requests on GitHub in the same fashion as the last module to review this exercise submission.

If you experience issues, ask a tutor for help!

Exercise

Part 1: Create a Virtual Machine

Vagrant provides a CLI tool to manage all VM-related options. This is preferable to using the GUI of VirtualBox (for example), as CLI tools are quicker to run and can be automated much more easily.

To start, navigate to the root of your code repository and run `vagrant init`. This will create a a file called "Vagrantfile" in the root of your repository. It's filled with some automatically-generated documentation which you can read if you like. When you're ready, we can trim this file down to the bare essentials. It should look like:

```
Vagrant.configure("2") do |config|
  config.vm.box = "hashicorp/bionic64"
end
```

This file specifies the **box** we're using. A box is a virtual machine image that we'll use as our starting point. `hashicorp/bionic64` is the image name - this simple image is provided by Hashicorp (the makers of Vagrant) and is suitable for our needs. You can browse other boxes online for more specific needs. A box includes the operating system, plus a variety of pre-installed software to reduce our own setup needs.

Vagrant also creates a `.vagrant` sub-directory. This includes specific details for your VM setup. You should add `.vagrant` to your `.gitignore` file as it is machine-specific. The `Vagrantfile` itself must be checked into git.

Part 2: Explore your virtual machine

The minimal Vagrantfile created in Part 1 is enough to get started. Open a terminal at the root of your code repository. Launch your VM and connect to it using a secure shell (SSH):

```
$ vagrant up
$ vagrant ssh
```

`vagrant ssh` is a convenient shortcut to SSH into the virtual machine. By default, it logs you in as the `vagrant` user. You can check your current user with the `whoami` builtin command.

Within the SSH session you can explore this Ubuntu virtual machine using the bash shell tools discussed throughout this module. For example, you can run `python --version` and `python3 --version` to discover the system Python versions pre-installed on the box.

Navigate to the `/vagrant` directory at the root of the virtual machine filesystem. You'll see that it contains all your app source code! The directory is mirrored from the host computer file system.

You can try to run your application as usual, but will notice that it fails. That's because the virtual machine's Python setup is not quite equivalent to the host machine and there's a few extra setup steps needed.

Rather than do it manually via your SSH session, we'll modify the Vagrantfile to automate the environment configuration. You can exit the SSH session via the keyboard shortcut `Ctrl/⌘ + D`.

Don't forget to commit!

Make sure you use **`git commit`** to save your work regularly.

It's good practice, and lets you revert back to a previous revision if you find you've broken everything horribly!

Try to make your commit messages concise and descriptive. Using source control tools effectively is just as important as the code you submit.

The VM can be managed using vagrant's CLI commands. Some useful ones are:

- `vagrant up` - Starts your VM, creating and provisioning it automatically if required.
- `vagrant provision` - Runs any VM provisioning steps specified in the Vagrantfile. Provisioning steps are one-off operations that adjust the system provided by the box.
- `vagrant suspend` - Suspends any running VM. The VM will be restarted on the next `vagrant up` command.
- `vagrant destroy` - Destroys the VM. It will be fully recreated the next time you run `vagrant up`.

Part 3: Automate Python configuration

We need to install an up-to-date version of Python rather than relying on the system version. We also need to configure our dependency manager (poetry) and finally install our dependencies.

To manage your Python environment, you'll use pyenv. Pyenv is a popular tool for managing multiple python versions on a single machine. This installation should be a **provisioning** script in the Vagrantfile. That way, vagrant will handle running the script when a new VM is created.

Add the following outline section to your Vagrantfile (within the configuration block defined earlier). This uses heredoc syntax to embed a shell script in the Vagrantfile:

```
config.vm.provision "shell", privileged: false, inline: <<-SHELL
  sudo apt-get update

  # TODO: Install pyenv prerequisites
  # TODO: Install pyenv
SHELL
```

The provisioning step is run when vagrant creates a new VM. It can also be forcibly re-run on an existing VM using `vagrant provision` or `vagrant up --provision`. By default, Vagrant would execute your script as `root`. By setting `privileged: false` the provisioning script instead runs as the `vagrant` user. It is convenient to use the same user for provisioning and SSH sessions. Doing so avoids potential differences in permissions and environment.

Once you've copied this template, check that it runs.

- **Install pyenv prerequisites**. The ubuntu system lacks some key development dependencies necessary to build more complex python libraries and extensions. Modify your vagrantfile to use `apt-get` to install the prerequisites listed here.
- **Install pyenv**. Follow the instructions on github for a basic git checkout.
    - Take care - the details are system-specific. You'll need to modify some parts.
    - To ensure `pyenv` is available to both vagrant and you (via SSH), _make sure the `PYENV_ROOT` and `PATH` environment variable modifications are made in the `~/.profile` file._

To test your `pyenv` installation, re-run your provisioning step then run the `pyenv` command via SSH. If successful you will be presented with usage instructions.

- **Use `pyenv` to install a custom Python version**. Download a new Python version using `pyenv install <version>` where is a Python version label (e.g. `3.8.5`). Make sure to pick a version that meets your app requirements!
- **Set a global Python version**. Use `pyenv global <version>` to make your new Python version the default one (likely replacing the end-of-life Python 2.7). This step isn't required for your app, but is a nice-to-have and smooths over some potential issues in the poetry installation process.

Re-provision your VM, and SSH into it. Check the default Python version with `python --version` - it should match the one you installed with pyenv!

Finally, you can download and install poetry by adding the following command to your provisioning script:

```
curl -sSL https://raw.githubusercontent.com/python-poetry/
poetry/master/get-poetry.py | python
```

Take a moment to review your provisioning setup. You should now have a Vagrantfile that any developer can use to create a virtual machine configured with a complete Python environment and pre-installed with poetry for managing virtual environments and code dependencies.

Part 4: Test it Works!

At this stage, you should check the VM can run your app!

1. Make sure you're VM is up-to-date with your latest provisioning code (destroy and re-create it if you want a clean slate).
2. SSH into the virtual machine
3. Navigate to your app code: `cd /vagrant`
4. Install your dependencies: `poetry install`
5. Launch flask's development server: `poetry run flask run`

If successful, you'll see the now-familiar flask development server startup logging, but you won't be able to access it in the browser - don't worry about that for now. We'll come back to networking once we've automated this step.

Troubleshooting

If this step fails it may be because poetry, running inside the VM, has automatically detected the host system virtual environment (located in the `.venv` directory) and is trying to use it instead of creating its own! Unfortunately there's no way to exclude the host virtual environment from the directory sharing using VirtualBox.

If you're having this issue, delete `.venv` and edit your `poetry.toml` file to set the `virtuals.in-project` option to `false`:

```
[virtualenvs]
create = true
in-project = false
```

This tells poetry to create virtual environment files in a central location, outside your code directory. This side-steps the issue, and allows the host machine and VM to run their own independent virtual environments.

Part 5: Automatically launch the app

The final step is to make vagrant automatically launch the app when you run `vagrant up`. Vagrant supports triggers which we can use for this.

Write a trigger to install your dependencies and launch the app after `vagrant up`, automating the steps you performed manually in Part 4. When done, you'll have a Vagrantfile that looks like:

```
Vagrant.configure("2") do |config|

  config.vm.box = "hashicorp/bionic64"

  config.vm.provision "shell", inline: <<-SHELL
    # Your provisioning script
  SHELL

  config.trigger.after :up do |trigger|
    trigger.name = "Launching App"
    trigger.info = "Running the TODO app setup script"
    trigger.run_remote = {privileged: false, inline: "
      # Install dependencies and launch
      # <your script here>
    "}
  end
end
```

It's important that you install dependencies as part of the trigger, rather than at the provisioning stage. The provisioning step won't be re-run unless you force it, so is best suited for one-time setup tasks. Dependencies may change.

Once this is ready, try launching your VM again with `vagrant up`. You should see console output similar to that below:

```
    default: Installing collected packages: itsdangerous, Werkzeug,
MarkupSafe, Jinja2, click, Flask, python-dotenv, urllib3, certifi,
chardet, idna, requests
    default: Successfully installed Flask-1.1.1 Jinja2-2.11.2
MarkupSafe-1.1.1 Werkzeug-1.0.1 certifi-2020.6.20 chardet-3.0.4
click-7.1.2 idna-2.10 itsdangerous-1.1.0 python-dotenv-0.12.0
requests-2.23.0 urllib3-1.25.10
    default:  * Serving Flask app "app" (lazy loading)
    default:  * Environment: development
    default:  * Debug mode: on
    default:  * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
    default:  * Restarting with stat
    default:  * Debugger is active!
    default:  * Debugger PIN: 828-965-351
```

Congratulations, your app is now running attached to the SSH session! Unfortunately you're not quite done. Navigating to `http://127.0.0.1:5000/` or `http://localhost:5000/` will not show you the app. Let's fix this networking issue.

Part 4: Networking

By default, flask's development server binds to the localhost interface (`127.0.0.1`) on port 5000, which is what you've been using so far. This works fine for local development, but won't work inside your VM (the VM's localhost is not the same as the host machine's). Let's walk through the steps needed to make your app, running inside a local VM, accessible via a browser on the host machine.

Firstly, redirect traffic from port 5000 on the host machine to port 5000 on the VM. This will let you access the app running in vagrant via your web browser. **Add the necessary**

configuration to your Vagrantfile (hint: have a look at the port forwarding documentation).

Second, you'll need to change the default host used by flask's development server. `127.0.0.1` won't work here. Instead, change your `flask run` command to specify `0.0.0.0` as the host (hint: `flask run` has a `--host` option you can use). This will let all network interfaces access the app, including the traffic from the mapped port on your host machine.

With these changes in place, re-launch your VM and check you can access your site by opening http://localhost:5000. The server will run attached to that SSH session (so leave the terminal open), and will hot reload with any new changes.

Congratulations! You've now got a working VM configuration that can easily be shared with other developers. With a single command (`vagrant up`) others can run your application using flask's development server, and browse the web app using a browser of their choice.

Stretch Goal: Production Configuration

In a production environment you would never use `flask run`. Instead, flask needs a WSGI application to sit between it and a production-ready web server (e.g. apache or nginx). A popular option here is gunicorn. While vagrant is usually used to share development environments, virtual machines are also a popular option for previewing production settings.

- Edit your vagrantfile to use gunicorn instead of flask's development server.
- Use gunicorn's daemon mode to run the application in the background, so it can persist beyond your SSH session.
- Write log events from gunicorn to a file on your host machine, so you can easily view them even while gunicorn is running in the background.

Stretch Goal: Provision as `root`

So far your provisioning scripts have run as the `vagrant` user, which is suitable for this exercise. However, some use-cases might use the `root` user for provisioning. This can help if you need to access system files, maintain separate environment configuration and generally reduce the amount of `sudo` commands required!

Restructure your Vagrantfile to provision the VM using the root user. The app should still run as `vagrant`.

Hints:

Issues with the Hypervisor

Getting vagrant working with your hypervisor can be problematic at times. Here are some potential fixes (on Windows)

VirtualBox

If you're encountering networking/download related issues (e.g. with checksums) then you'll want to make sure that the `Virtual Machine Platform`, `Windows Subsystem for Linux` & `Hyper-V` features are turned off (this will require a restart).

- Keep in mind this will prevent you from using `docker` in the next module so you'll need to turn these features back on when you've finished (alternatively you could use Hyper-V as the hypervisor)

Hyper-V

Running `vagrant up` with Hyper-V requires more manual steps than using virtualbox:

1. You'll need to choose a virtual network switch to use (see here for instructions on how to do this)
2. You'll need to provide credentials for mounting the project folder as `/vagrant` on the vagrant box.
    - When prompted you should provide your username in the format `{Username}@{Domain}` (note `@` and not `\`).
3. There are significant limitations with controlling the networking configuration using Hyper-V. You'll need to note the IP address generated by Hyper-V when running `vagrant up`. One upside is that you don't have to forward the port over in the vagrantfile.

Writing the Configuration Script

- Configuration scripts can often get confusing! Use comments to explain what you're doing, and use whitespace to group blocks of related setup steps. In the examples above we've used heredoc syntax to keep all the setup code in the Vagrantfile, but you can also create separate script files and reference them from the Vagrantfile if you prefer.

- The authors of `pyenv` provide a useful reference page on UNIX shell initialisation. This may help if you're getting confused between `.bash_rc`, `.bash_profile`, `.profile` and others!

- Installing `pyenv` is a multi-step process, and requires some adjustment of your shell environment. If you're struggling to understand how to approach this provisioning step, take a look at the provisioning snippet below. Try it out - can you understand what each command does?

```
config.vm.provision :shell, privileged: false, inline: <<-SHELL

  ...

  git clone https://github.com/pyenv/pyenv.git ~/.pyenv
  echo 'export PYENV_ROOT="$HOME/.pyenv"' >> ~/.profile
  echo 'export PATH="$PYENV_ROOT/bin:$PATH"' >> ~/.profile
  echo 'eval "$(pyenv init -)"' >> ~/.profile
  source ~/.profile

  ...

SHELL
end
```

- `flask run` automatically loads the `.env` file into environment variables which are read by your flask app. `gunicorn` does not, and you'll need to write this yourself.