Vagrant

Vagrant is used to create VMs on the local machine which mirror the production environment as closely as possible.

Currently, there is one master VM, at least one Worker VM, and one testing VM in production. This architecture is mirrored by Vagrant on the local machine.

Starting Vagrant

```
vagrant up
```

This is the command to start the VMs for the first time. It will read the Vagrant file in the root directory and create all the necessary VMs and also provision them.

```
  config.vm.synced_folder ".", "/titb",
    owner: "vagrant",
    group: "www-data",
    mount_options: ["dmode=777,fmode=777"]
```

These lines ensure that the project is copied into the VMs and also synchronised. This is convenient as it enables the developer to make changes in the code on the local machine which is immediately reflected in the code within the VMs.

The Ansible Playbook that is being used to provision the machines is specified here:

```
    ansible.playbook          = "provisioning/vagrant.yml"
```

It is also possible to use a different playbook for provisioning.

Reprovisioning After Changes to the Code

TODO: mention the deploy_container tags

Once the code has been updated locally, it is not necessary to restart the entire VM. You can reprovision the VMs by running:

```
vagrant provision
```

If the only code that has changed only concerns e.g. the master VM, you can specify to only provision this VM with:

```
vagrant provision master
```

This will also work with the Worker (here use `worker1`, `worker2`, ...) and the testing VM. Specifying the exact VMs will work with every vagrant command which is mentioned in this documentation.

Ansible has a tagging system and if you only want to provision specific Ansible rules within a tag this is also possible. For example, to only restart the Docker containers uncomment the following line:

```
    ansible.tags                 = "deploy_container"
```

If you made changes to the Vagrant file outside of the `config.vm.provision "ansible" do |ansible|` scope you need to run:

```
vagrant reload
```

to reflect the changes in the VMs. This can be useful e.g. when you want to turn off the `config.vm.synced_folder` option and delete the required lines, but do not want to tear down and restart the entire VM (which usually takes long). However, most of the time the command `vagrant provision` does the necessary changes to the VM.

Debugging a Vagrant VM

To debug a VM or access the logs you ssh into the VMs by running:

```
vagrant ssh <VM-name>
```

Destroying Vagrant VMs

To destroy the vagrant VMs simply run

```
vagrant destroy -f
```

Sometimes it is worth to destroy the VM and restarting it completely to check for bugs before pushing to production.

Increasing the Amount of Worker VMs

The biggest computational bottleneck in the Klee Web application is running the KLEE application on the submitted code. This is managed by the Workers which can be scaled up and down flexibly in production.

There is two ways to scale the amount of Workers:

- Increase the amount of Worker VMs
- Increase the number of running Worker Docker containers within each Worker VM.

Changing the amount of Worker VMs

In the development environment this is simply done by changing this line in the Vagrantfile:

```
WORKER_COUNT = 1
```

Then the new Workers need to be created with `vagrant up` as described above. Once provisioned, they will automatically register themselves without any changes to the Master VM.

In production you would need to create the VM and provision it as well. Please refer to the DEPLOY.md documentation.

Changing the amount of Worker VMs

Alternatively to creating new VMs, you can also try out to have more Worker instances within each VM. This will not increase the overall computing power, but the concurrency will be increased.

To do that you need to change the provisioning rule for the Worker within `/provisioning/roles/worker/tasks/deploy.yml`.

```
- name: Start docker Worker
  docker_container:
    name: "worker-{{ item }}"
    image: celery_worker
    restart_policy: always
    volumes:
    # can start other containers:
    - /var/run/docker.sock:/var/run/docker.sock
    # can place results into /tmp/ folder
    - /tmp/:/tmp/
    # connect source code paths
    - "{{ klee_env_path_etc }}:{{ klee_env_path_etc }}"
    - "{{ python_runner }}:{{ python_runner }}"
    - "{{ worker_dir }}:{{ worker_dir }}"
    network_mode: "{{ 'host' if (ci) else 'bridge' }}"
    recreate: yes
  with_sequence: start=1 end=2
```

Here to scale the running instances of Workers change the `end=2` to any positive number greater or equal than 1.

Test It Out!

You can quickly test what difference it makes to have one Worker or multiple instances of Workers running. Open the website at http://192.168.33.10 in multiple tabs. Run the `regexp.c` example on all the tabs at the same time with only one Worker running on one Worker VM only.

Then, try to either create more Worker VMs or run more Worker instances per Worker VM and repeat the test. You should notice an increase in parallel execution of the task.

Provisioning Like in the Production Environment

You can create VMs with the same Ansible Playbook, environmental variables, and secrets as in the production environment. To do so comment out the folder synchronisation

```
  # config.vm.synced_folder ".", "/titb",
  #   owner: "vagrant",
  #   group: "www-data",
  #   mount_options: ["dmode=777,fmode=777"]
```

Use the production Ansible Playbook for provisioning and require to ask for the Ansible Vault password.

```
      ansible.playbook          = "provisioning/prod.yml"
      ansible.ask_vault_pass    = true
      # ansible.playbook          = "provisioning/vagrant.yml"
```

It is best to destroy the VM and restart them

```
vagrant destroy -f
vagrant up
```

You will be prompted for the Ansible Vault password, which you can take from the `~/.klee_vault_password` file in your root directory.

Doing this can give you additional security to test the application before pushing the code and restarting the production environment.

Please note that if you work on your own fork you need to have all the updates pushed onto your master branch and also change the variable `git_repo` within the `/provisioning/group_vars/all` file to specify the URL to your GitHub fork.