LS lab 2 - Infrastructure as Code (IaC)

> A team of two assignment

The lab goal is to understand and learn the core tools and techniques that are used in day by day DevOps routine.

Task 1 - IaC Theory

Briefly answer for the following questions what is and for what:

- **git** repository:
  - `/.git`
  - `/.github`
  - `.gitignore`
  - `.gitmodules`
- **ansible** directory:
  - `ansible.cfg`
  - `inventory` folder
  - `roles` folder
    - `tasks`
    - `defaults/group_vars`
    - `handlers`
    - `templates`
  - `playbooks` folder
- **terraform** folder:
  - `main.tf`
  - `variables.tf`
  - `outputs.tf`

Task 2 - Prepare your application

Find and choose (it is much better to develop by yourself) a simple application. For example, it could be a web server with the static HTML page, time zones server or a currency calculator. Use whatever programming language that you want (python, golang, C#, java...). Include the link to VCS where your application is stored.

Bonus: prepare microservices instead of standalone application, e.g. full stack web application with web server, database...

Task 3 - Dockerize your application

1. Build Docker image for your application (make *Dockerfile*).

   Look for the best *Dockerfile* practices and try to follow them.

Bonus: use docker-compose in the case of microservices.

Task 4 - Deliver your application using Software Configuration Management

1. Get your personal cloud account. Free tiers for a AWS and GCP users has been blocked in Russia. If you already have accounts, it should work and be enough for this lab. If not, try other cloud providers with a free subscription: Yandex.Cloud, IBM, Oracle, Alibaba Cloud... If you will not be able to work with cloud, you have to proceed within the local deployment for the whole Task 4. For example, prepare a local virtual machine for the further tasks. Include the explanation into the report why you were forced to work locally.
2. Use Terraform to deploy your required cloud instance. Please notice that to run `terraform init` command you have to use VPN.

   Look for the best Terraform practices and try to follow them.

   If for a some reason you will not able to use VPN, prepare a local VM using Vargant tool. Include the explanation into the report about the inability to work with VPN.
3. Choose Software Configuration Management (SCM) tool. Ansible is the **industry standard**. Of course, we have other SCM solutions such as SaltStack, Puppet, Chef. You can try them but remember that it is probably more difficult to work with these tools and you are responsible for your choice.
4. Using SCM tool, write a playbook tasks to deliver and run your application to cloud instance/to local VM. *Try to separate your configuration files into* ***inventory/roles/playbooks*** *files. In real practice, we almost newer use poor playbooks where everything all in one.*

   Also try to use the best practices for you SCM tool, e.g. Ansible.

   Bonus: use Ansible Molecula and Ansible Lint to test your application before to deliver it to cloud.

   Bonus: Deploy Ansible AWX and demonstrate a PoC.

Task 5 - Teamwork with the version control system

In industry, before deploying a new version of the service/application, we receive a review from colleagues.

1. a) Create and log in to your personal version control system account on the `git` engine:

- github (**default choice**)
- gitlab
- bitbucket
- ...

b) Create a repository with your application/microservices and all required code/configs.

c) Synchronize your local and remote repository.

d) Create a separate branch for the development, as well as protect your master branch in the repository from direct commits to it.

e) Create a Pull Request from your developer branch to the master branch.

f) Your colleague should get explanation about your work and conduct a review of your PR.

g) Receive an PR approvement, merge your PR and synchronize the local and remote repositories.

Bonus: implement steps 2 and 4 using Terraform.

2. Learn some advanced `git` features. Answer for the following questions and show a PoC wherever it is possible for you:

   a) What's the difference between `git pull` and `git fetch`?

   b) What's the difference between `git rebase` and `git merge`?

   c) How we can `Rebase` one branch with commits from other head branch? And to *replace* one branch by another?

   d) How we can *replace (overwrite)* one branch with other remote upstream branch entirely?

   e) How we can add a `remote-tracking` repository? When it's suitable?

   f) How we can *merge* fork branch with origin/upstream branch?

   g) How we can *push* the new branch to origin when this branch is set up *for tracking to the specified remote-tracking* repository?

   h) How we can *join* several commits into one?

   i) How we can *change* commit message?

   j) How we can `cancel(undo)` the last pushed commit to the target remote branch?

Bonus: learn and try to use in practice `git hooks`. For example, we might use hooks to prevent commits pushing which have incorrect message body that does not comply with the repository policy.

As the final result after applying all configurations files, you should be able to show the working execution of your running app on the cloud instance (local VM) according to this app destiny. For example, if you prepared a web server, you have to be able to open http://yourdomain.com in your browser and see the web site page.

Bonus: Play with SCM cases

The goal is to easily, reliably and quickly maintain different kinds of systems at once. Prepare two different guest systems (in cloud or as local VMs) e.g. an Ubuntu Server and Fedora. Then play with SCM tool to automate system preparations on those. You might create and provide your own ideas or follow to some suggestions:

- NTP
- set the default shell
- Apache Web Server
- Docker deployment
- Kubernetes namespace deployment
- Nginx & PHP-FPM & MariaDB & WordPress
- Ansible Vault (definitely good choice)
- IPtables (close all machine ports exclude ssh, http, https)
- development environments organization (ssh keys management...)
- ...

Never forget to use best practices. Complete as many tasks as possible.

Appendix 1 - Some Docker best practices

- Do not use `:latest`

- Exclude with .dockerignore
- Minimize the number of layers
- Build the smallest image possible
- Follow tag versioning of your images
- Don't be a root user

Appendix 2 - Some Terraform best practices

- Make understandable naming
- Don’t commit the `.tfstate` file
- Back up state files
- Manipulate state only through commands
- Use variables
- Use validate and plan commands frequently
- Abstract code to maximize reuse
- Use modules where it is necessary