

1

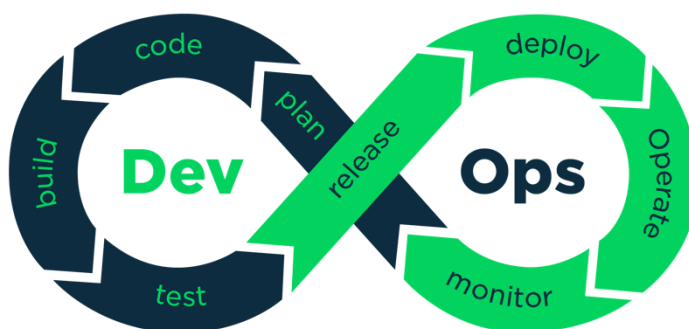
Lab

PHỤC VỤ MỤC ĐÍCH GIÁO DỤC
FOR EDUCATIONAL PURPOSE

AUTOMATING EVERYTHING AS CODE

DevOps/CI-CD/Git

Thực hành Lập trình An toàn & Khai thác lỗ hổng phần mềm



Lưu hành nội bộ

<Ng nghiêm cấm đăng tải trên internet dưới mọi hình thức>

A TỔNG QUAN

A.1 Mục tiêu

- Giới thiệu

A.2 Thời gian thực hành

- Thực hành tại lớp: **5** tiết tại phòng thực hành.
- Hoàn thành báo cáo kết quả thực hành: tối đa **13** ngày.

A.3 Môi trường thực hành

Sinh viên cần chuẩn bị trước máy tính với môi trường thực hành như sau:

1 PC cá nhân với hệ điều hành tự chọn, trong đó có 1 máy ảo **Linux** cài đặt Docker (hoặc Windows Subsystem Linux version2)

A.4 Các tài nguyên được cung cấp sẵn

- Dành cho Phần B.2: source ứng dụng **sample-app.zip**.
- Dành cho Phần B.3: **unittest.tar.gz**

B THỰC HÀNH

B.1 Quản lý phiên bản phần mềm với Git

Yêu cầu 1. Sinh viên thực hiện các bước bên dưới để thiết lập hệ thống quản lý phần mềm với Git trên máy ảo, báo cáo kết quả của các bước và trả lời các câu hỏi.

B.1.1 Thiết lập Git Repository

Bước 1. Cấu hình thông tin người sử dụng và liên kết tài khoản trong local repository.

Lưu ý: sử dụng user.name là **tên nhóm sinh viên**, ví dụ “Nhóm 01” và email của đại diện nhóm.

```
$ git config --global user.name "<username>"
$ git config --global user.email <email>
```

Ví dụ:

```
ubuntu@ubuntu: ~
ubuntu@ubuntu:~$ git config --global user.name "Nhóm 00"
ubuntu@ubuntu:~$ git config --global user.email 13520260@gm.uit.edu.vn
```

Bước 2. Kiểm tra lại thông tin người sử dụng với lệnh

```
$ git config --list
```

```
ubuntu@ubuntu: ~
ubuntu@ubuntu:~$ git config --list
user.name=Nhóm 00
user.email=13520260@gm.uit.edu.vn
ubuntu@ubuntu:~$
```

Bước 3. Tạo thư mục có tên **NhómX**, với X là số thứ tự nhóm ở dạng 2 chữ số, và di chuyển đến thư mục vừa tạo

```
$ mkdir NhómX && cd NhómX
```

Bước 4. Trong thư mục **NhómX**, tiếp tục tạo thư mục **git-intro** và di chuyển vào

```
$ mkdir git-intro && cd git-intro
```

Đây là thư mục sẽ được sử dụng làm Git repository cục bộ trên máy sinh viên.

```
ubuntu@ubuntu: ~/Nhóm00/git-intro
ubuntu@ubuntu:~$ mkdir Nhóm00 && cd Nhóm00
ubuntu@ubuntu:~/Nhóm00$ mkdir git-intro && cd git-intro
ubuntu@ubuntu:~/Nhóm00/git-intro$
```

Bước 5. Khởi tạo thư mục hiện tại (**git-intro**) dưới dạng Git repository với lệnh:

```
$ git init
```

Output của lệnh trên cho biết ta đã tạo một local repository được chứa trong thư mục **.git**. Đây là nơi chứa tất cả lịch sử thay đổi của code.

Có thể xem lại các thư mục đã được tạo sau lệnh **git init** bằng lệnh:

```
$ ls -a
```

```
ubuntu@ubuntu: ~/Nhom00/git-intro
ubuntu@ubuntu:~/Nhom00/git-intro$ git init
Initialized empty Git repository in /home/ubuntu/Nhom00/git-intro/.git/
ubuntu@ubuntu:~/Nhom00/git-intro$ ls -a
.  ..  .git
ubuntu@ubuntu:~/Nhom00/git-intro$
```

Bước 6. Xem trạng thái của repository cục bộ với lệnh

```
$ git status
```

Lệnh này cho phép kiểm tra các tệp nào đã bị thay đổi, hữu ích khi ta làm việc trong dự án và chỉ muốn commit một vài tệp tin chứ không phải toàn bộ.

```
ubuntu@ubuntu: ~/Nhom00/git-intro
ubuntu@ubuntu:~/Nhom00/git-intro$ git status
On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
ubuntu@ubuntu:~/Nhom00/git-intro$
```

Output của lệnh git status hiển thị các tệp tin đã sửa đổi trong thư mục, được chuẩn bị cho lần commit sắp tới. Ở ví dụ trên, một số thông tin hiển thị bao gồm:

- Nhánh (branch) đang làm việc: master
- Chưa có commit nào, commit này là Initial commit (lần đầu)
- Không có gì thay đổi trong commit.

B.1.2 Staging và Committing một tập tin trên Repository

Bước 1. Trong thư mục **git-intro**, tạo 1 tệp tin tên **README.MD**, có nội dung là tên nhóm và danh sách thành viên của nhóm.

Bước 2. Kiểm tra tệp tin vừa tạo.

Kiểm tra tệp tin đã được tạo trong thư mục với lệnh:

```
$ ls -la
```

```
ubuntu@ubuntu: ~/Nhom00/git-intro
ubuntu@ubuntu:~/Nhom00/git-intro$ ls -la
total 16
drwxrwxr-x 3 ubuntu ubuntu 4096 Sep 11 08:50 .
drwxrwxr-x 3 ubuntu ubuntu 4096 Sep 11 08:47 ..
drwxrwxr-x 7 ubuntu ubuntu 4096 Sep 11 08:48 .git
-rw-rw-r-- 1 ubuntu ubuntu  76 Sep 11 08:50 README.MD
ubuntu@ubuntu:~/Nhom00/git-intro$
```

Kiểm tra nội dung của tệp tin với lệnh:

```
$ cat README.MD
```

```
ubuntu@ubuntu: ~/Nhom00/git-intro
ubuntu@ubuntu:~/Nhom00/git-intro$ cat README.MD
Nhom 00
20520xxx - Tran Van A
20520yyy - Bui Thi B
20520zzz - Nguyen Ngoc C
ubuntu@ubuntu:~/Nhom00/git-intro$
```

Bước 3. Kiểm tra trạng thái Repository sau khi tạo tập tin.

```
$ git status
```

Mình chứng cho thấy Git đã tìm thấy 1 tập tin mới nhưng chưa giám sát?

Bước 4. Staging tập tin – Đưa tập tin mới vào vùng staging

Trước khi commit, tập tin cần được đưa vào vùng staging, sử dụng lệnh

```
$ git add README.MD
```

Lệnh này tạo một snapshot cho tệp. Mọi thay đổi của tệp này sẽ được ghi nhận để chuẩn bị commit.

Bước 5. Kiểm tra trạng thái Repository sau khi ‘stage’ tập tin.

```
$ git status
```

Mình chứng cho thấy Git thấy file mới README.MD trên stage?

Bước 6. Commit tập tin

Đối với các thay đổi đã được đưa vào vùng ‘staging’, cần thực hiện tiếp bước Commit để Git thay đổi những thay đổi đó, với lệnh git commit như sau:

Lưu ý: thay NhomX tương ứng với nhóm.

```
$ git commit -m "Committing README.MD from NhomX to begin tracking changes"
```

Trong đó:

- Tuỳ chọn -m cho phép thêm thông điệp giải thích những thay đổi đang thực hiện.
- Lưu ý số và chữ được highlight là commit ID. Mọi commit được xác định bằng một hàm băm SHA1 duy nhất. Commit ID là 7 ký đầu tiên trong commit hash.

```
ubuntu@ubuntu: ~/Nhom00/git-intro
ubuntu@ubuntu:~/Nhom00/git-intro$ git commit -m "Committing README.MD fr
om Nhom00 to begin tracking changes"
[master (root-commit) 3680d92] Committing README.MD from Nhom00 to begin
tracking changes
1 file changed, 4 insertions(+)
create mode 100644 README.MD
ubuntu@ubuntu:~/Nhom00/git-intro$
```

Bước 7. Xem lịch sử commit

```
$ git log
```

Lệnh này hiển thị tất cả các commit trong lịch sử của nhánh hiện tại. Theo mặc định, tất cả các commit là được thực hiện cho nhánh master.

```
ubuntu@ubuntu: ~/Nhom00/git-intro
ubuntu@ubuntu:~/Nhom00/git-intro$ git log
commit 3680d92243b0209b1971d54bf04eef4bd11e0aac (HEAD -> master)
Author: Nhom 00 <13520260@gm.uit.edu.vn>
Date:   Sun Sep 11 08:54:01 2022 -0700

    Committing README.MD from Nhom00 to begin tracking changes
ubuntu@ubuntu:~/Nhom00/git-intro$
```

B.1.3 Sửa đổi tập tin và theo dõi các thay đổi

Bước 1. Sửa đổi tập tin README.MD

Chèn thêm 1 dòng bất kỳ vào cuối tập tin README.MD, sử dụng lệnh **echo** và dấu ">>" như bên dưới.

Lưu ý: Dùng nhầm dấu ">" sẽ ghi đè lên tập tin hiện có.

```
$ echo "I am beginning to understand Git" >> README.MD
```

Bước 2. Xem lại nội dung tập tin đã chỉnh sửa.

```
$ cat README.MD
```

```
ubuntu@ubuntu: ~/Nhom00/git-intro
ubuntu@ubuntu:~/Nhom00/git-intro$ echo "I am beginning to understand Git" >> README.MD
ubuntu@ubuntu:~/Nhom00/git-intro$ cat README.MD
Nhom 00
20520xxx - Tran Van A
20520yyy - Bui Thi B
20520zzz - Nguyen Ngoc C
I am beginning to understand Git
ubuntu@ubuntu:~/Nhom00/git-intro$
```

Bước 3. Kiểm tra thay đổi đối với repository

```
$ git status
```

Mình chứng cho thấy Git đã thấy các thay đổi mới chưa được commit?

Bước 4. Stage và commit tập tin đã thay đổi

Sử dụng các lệnh:

```
$ git add README.MD
$ git commit -m "NhomX Added additional line to file"
```

Git đưa ra các output như thế nào để hiển thị các thay đổi trong tập tin vừa commit?

Bước 5. Xem lại commit vừa thực hiện trong repository

```
$ git log
```

```
ubuntu@ubuntu: ~/Nhom00/git-intro
ubuntu@ubuntu:~/Nhom00/git-intro$ git log
commit 26e0a0c7dc35afc054d67a503fbc72e0053dabe5 (HEAD -> master)
Author: Nhom 00 <13520260@gm.uit.edu.vn>
Date:   Sun Sep 11 08:56:58 2022 -0700

    Nhom00 Added additional line to file

commit 3680d92243b0209b1971d54bf04eef4bd11e0aac
Author: Nhom 00 <13520260@gm.uit.edu.vn>
Date:   Sun Sep 11 08:54:01 2022 -0700

    Committing README.MD from Nhom00 to begin tracking changes
ubuntu@ubuntu:~/Nhom00/git-intro$
```

Cho biết thông tin về commit vừa thực hiện: Commit ID, thời gian commit, thông điệp commit?

Bước 6. So sánh các commit. Khi có nhiều commit trong log, có thể so sánh khác biệt giữa hai commit bằng lệnh:

```
$ git diff <commit ID 1> <commit ID 2>
```

Sinh viên thử so sánh 2 commit đã tạo, giải thích ngắn gọn ý nghĩa các ký hiệu trong output?

```
ubuntu@ubuntu: ~/Nhom00/git-intro
ubuntu@ubuntu:~/Nhom00/git-intro$ git diff 3680d92 26e0a0c
diff --git a/README.MD b/README.MD
index a7f8f07..aa208a2 100644
--- a/README.MD
+++ b/README.MD
@@ -2,3 +2,4 @@ Nhom 00
 20520xxx - Tran Van A
 20520yyy - Bui Thi B
 20520zzz - Nguyen Ngoc C
+I am beginning to understand Git
ubuntu@ubuntu:~/Nhom00/git-intro$
```

B.1.4 Branches và Merging (nhánh và hợp nhất)

a) Làm việc trong branch

Khi một repository được tạo ra, tất cả các tập tin sẽ được đưa vào một nhánh chính là master. Việc phân nhánh sẽ giúp kiểm soát và thực hiện thay đổi trong một vùng mà không hưởng nhánh chính, tránh code bị ghi đè không mong muốn.

Bước 1. Tạo branch mới

Tạo branch mới **feature** với câu lệnh:

```
$ git branch feature
```

Bước 2. Kiểm tra branch hiện tại

```
$ git branch
```


Lệnh git branch không kèm theo tên branch sẽ hiển thị tất cả các branch cho repository này. Trong đó, nhánh có ký hiệu dấu * phía trước là nhánh hiện tại đang làm việc (đang checkout).

```
ubuntu@ubuntu: ~/Nhom00/git-intro
ubuntu@ubuntu:~/Nhom00/git-intro$ git branch feature
ubuntu@ubuntu:~/Nhom00/git-intro$ git branch
  feature
* master
ubuntu@ubuntu:~/Nhom00/git-intro$
```

Bước 3. Chuyển (checkout) branch mới

Để chuyển sang nhánh mới, ví dụ nhánh **feature**, sử dụng lệnh:

```
$ git checkout <branch-name>
```

```
ubuntu@ubuntu: ~/Nhom00/git-intro
ubuntu@ubuntu:~/Nhom00/git-intro$ git checkout feature
Switched to branch 'feature'
```

Bước 4. Kiểm tra lại branch hiện tại

Chạy lại lệnh

```
$ git branch
```

Dấu hiệu nào cho thấy đã chuyển sang nhánh mới là feature?

Bước 5. Thay đổi tập tin README.MD, stage và git trên branch feature

Sinh viên thực hiện và báo cáo các bước:

- Thêm một đoạn text mới dạng "... from branch feature" vào tập tin README.MD.
- Stage tập tin vừa thay đổi trong branch feature.
- Kiểm tra trạng thái repository sau khi stage tập tin.
- Commit tập tin trong branch feature với thông điệp "Added a third line in feature branch".
- Xem lịch sử các commit đã thực hiện, ***minh chứng cho thấy có 1 commit tại nhánh feature?***

b) Merge các thay đổi từ các branch vào branch master

Bước 1. Chuyển (checkout) sang branch master

```
$ git checkout master
```

Bước 2. Kiểm tra file README.MD ở nhánh master để xem nội dung của nó có bị ảnh hưởng bởi thay đổi ở nhánh feature.

Bước 3. Merge tất cả các nội dung tập tin thêm từ branch feature sang master

Các branch thường được sử dụng khi triển khai các tính năng mới hoặc sửa lỗi. Các thay đổi có thể được gửi cho các thành viên trong nhóm phát triển xem xét và thống nhất, sau đó có thể gộp vào branch chính master.

Để merge nội dung từ 1 branch về master có thể sử dụng lệnh:

```
$ git merge <branch-name>
```

Trong đó, branch-name là branch muốn pull về branch hiện tại.

Báo cáo kết quả thực hiện câu lệnh merge branch feature về branch master?

```
ubuntu@ubuntu: ~/Nhom00/git-intro
ubuntu@ubuntu:~/Nhom00/git-intro$ git merge feature
Updating 26e0a0c..94e4c34
Fast-forward
 README.MD | 1 +
 1 file changed, 1 insertion(+)
ubuntu@ubuntu:~/Nhom00/git-intro$
```

Bước 4. Kiểm tra nội dung của file README.MD khi merge

Mở tập tin README.MD, so sánh nội dung của nó với kết quả sau lần thay đổi cuối ở branch master ở phần **Bước 2**?

```
ubuntu@ubuntu: ~/Nhom00/git-intro
ubuntu@ubuntu:~/Nhom00/git-intro$ cat README.MD
Nhom 00
20520xxx - Tran Van A
20520yyy - Bui Thi B
20520zzz - Nguyen Ngoc C
I am beginning to understand Git
Hello world from branch feature
ubuntu@ubuntu:~/Nhom00/git-intro$
```

Bước 5. Xoá branch

Kiểm tra branch feature vẫn còn tồn tại bằng lệnh

```
$ git branch
```

```
ubuntu@ubuntu: ~/Nhom00/git-intro
ubuntu@ubuntu:~/Nhom00/git-intro$ git branch
feature
* master
ubuntu@ubuntu:~/Nhom00/git-intro$
```

Có thể xoá branch bằng lệnh sau, trong đó <branch-name> là tên branch muốn xoá. Lưu ý, phải chuyển sang branch khác trước khi muốn xoá 1 branch, Git không cho phép xoá branch đang sử dụng.

```
$ git branch -d <branch-name>
```

Xoá branch feature và kiểm tra lại branch này còn tồn tại không?

B.1.5 Xử lý xung đột khi merge

Đôi khi sẽ có trường hợp xảy ra conflict (xung đột) khi merge. Ví dụ, khi ta thực hiện các thay đổi chồng chéo đối với tập tin, có thể Git sẽ không tự động merge những thay đổi này được.

Bước 1. Tạo branch test mới và chuyển sang branch test vừa tạo

```
$ git branch test
$ git checkout test
```

```
ubuntu@ubuntu: ~/Nhom00/git-intro
ubuntu@ubuntu:~/Nhom00/git-intro$ git branch test
ubuntu@ubuntu:~/Nhom00/git-intro$ git checkout test
Switched to branch 'test'
```

Bước 2. Chỉnh sửa nội dung tập tin README.MD với lệnh sau:

```
$ sed -i 's/feature/test/' README.MD
```

Giải thích ý nghĩa của dòng lệnh trên? Minh chứng sự thay đổi trước và sau khi chạy dòng lệnh?

```
ubuntu@ubuntu: ~/Nhom00/git-intro
ubuntu@ubuntu:~/Nhom00/git-intro$ sed -i 's/feature/test/' README.MD
ubuntu@ubuntu:~/Nhom00/git-intro$ cat README.MD
Nhom 00
20520xxx - Tran Van A
20520yyy - Bui Thi B
20520zzz - Nguyen Ngoc C
I am beginning to understand Git
Hello world from branch test
ubuntu@ubuntu:~/Nhom00/git-intro$
```

Bước 3. Thực hiện stage, commit branch test

```
$ git commit -a -m "Change feature to test"
```

Lưu ý: option -a chỉ ảnh hưởng đến các tập tin đã được sửa đổi và xóa. Nó không ảnh hưởng đến các tập tin mới.

```
ubuntu@ubuntu: ~/Nhom00/git-intro
ubuntu@ubuntu:~/Nhom00/git-intro$ git commit -a -m "Change feature to test"
[test 1bfc348] Change feature to test
1 file changed, 1 insertion(+), 1 deletion(-)
ubuntu@ubuntu:~/Nhom00/git-intro$
```

Bước 4. Chuyển sang branch master và thực hiện chỉnh sửa tập tin README.MD.

```
$ git checkout master
$ sed -i 's/feature/master/' README.MD
```

Nội dung tập tin README.MD thay đổi như thế nào?

```
ubuntu@ubuntu: ~/Nhom00/git-intro
ubuntu@ubuntu:~/Nhom00/git-intro$ git checkout master
Switched to branch 'master'
ubuntu@ubuntu:~/Nhom00/git-intro$ sed -i 's/feature/master/' README.MD
ubuntu@ubuntu:~/Nhom00/git-intro$ cat README.MD
Nhom 00
20520xxx - Tran Van A
20520yyy - Bui Thi B
20520zzz - Nguyen Ngoc C
I am beginning to understand Git
Hello world from branch master
```

Bước 5. Stage và commit branch master

```
$ git commit -a -m "Changed feature to master"
```

```
ubuntu@ubuntu: ~/Nhom00/git-intro
ubuntu@ubuntu:~/Nhom00/git-intro$ git commit -a -m "Change feature to master"
[master ea56d52] Change feature to master
1 file changed, 1 insertion(+), 1 deletion(-)
ubuntu@ubuntu:~/Nhom00/git-intro$
```

Bước 6. Merge hai branch test và master

```
$ git merge test
```

Kết quả không thể merge do có xung đột.

```
ubuntu@ubuntu: ~/Nhom00/git-intro
ubuntu@ubuntu:~/Nhom00/git-intro$ git merge test
Auto-merging README.MD
CONFLICT (content): Merge conflict in README.MD
Automatic merge failed; fix conflicts and then commit the result.
ubuntu@ubuntu:~/Nhom00/git-intro$
```

Bước 7. Tìm xung đột

- Xem các commit. Lưu ý rằng phiên bản HEAD là branch master. Cái này sẽ hữu ích trong bước tiếp theo.

```
$ git log
```

```
ubuntu@ubuntu: ~/Nhom00/git-intro
ubuntu@ubuntu:~/Nhom00/git-intro$ git log
commit ea56d52fea25752055be093a98afd94eadf7d1a4 (HEAD -> master)
Author: Nhom 00 <13520260@gm.uit.edu.vn>
Date: Sun Sep 11 09:18:46 2022 -0700

    Change feature to master
```

- Tập README.MD sẽ chứa thông tin tìm ra xung đột. Phiên HEAD (branch master) cho chứa từ “master” đang xung đột với phiên bản branch test là từ “test”.

```
ubuntu@ubuntu: ~/Nhom00/git-intro
ubuntu@ubuntu:~/Nhom00/git-intro$ cat README.MD
Nhom 00
20520xxx - Tran Van A
20520yyy - Bui Thi B
20520zzz - Nguyen Ngoc C
I am beginning to understand Git
<<<<<< HEAD
Hello world from branch master
=====
Hello world from branch test
>>>>>> test
ubuntu@ubuntu:~/Nhom00/git-intro$
```

Bước 8. Chỉnh sửa tập tin README.MD để xóa đoạn xung đột

Mở tập tên README.MD và xoá đoạn highlight với các ký hiệu <<, == và dòng được thêm ở branch test. Lưu lại và kiểm tra kết quả.

```
...
I am beginning to understand Git
<<<<<<< HEAD
... from branch master
=====
... from branch test
>>>>>> test
```

Bước 9. Thực hiện stage, commit và kiểm tra commit branch master

```
$ git add README.MD
$ git commit -a -m "Manually merged from test branch"
```

Kết quả commit? Kiểm tra log commit của branch master?

B.1.6 Tích hợp Git với Github

Hiện tại, những thay đổi chỉ được lưu trữ tại máy. Git chạy cục bộ và không yêu cầu bất kỳ máy chủ hoặc dịch vụ lưu trữ đám mây nào. Git cho phép người dùng lưu trữ cục bộ và quản lý tập tin.

Việc sử dụng Github sẽ giúp làm việc nhóm tốt hơn và tránh tình trạng mất dữ liệu. Có một số dịch vụ Git khá phổ biến bao gồm GitHub, Stash từ Atlassian và GitLab.

Yêu cầu 2. Sinh viên thực hiện các bước bên dưới để thiết lập hệ thống quản lý phần mềm với Git kết hợp Github, báo cáo kết quả của các bước.

- Bước 1: Tạo tài khoản Github
- Bước 2: Đăng nhập tài khoản Github
- Bước 3: Tạo Repository tên **devops-study-team** có chế độ Private
- Bước 4: Tạo thư mục **devops-study-team** trên máy ảo
- Bước 5: Sao chép tập tin **README.MD** vào thư mục vừa tạo
- Bước 6: Khởi tạo Git repository từ thư mục **devops-study-team**

Lưu ý: Cần điều chỉnh user.name và user.email cho khớp với tài khoản Github.

- Bước 7: Trỏ Git repository đến GitHub repository

Gợi ý: sử dụng lệnh **git remote add origin <GitHub link>**

- Bước 8: Stage, commit README.MD với thông điệp “Added to devasc-study-team”
- Bước 9: Kiểm tra log commit
- Bước 10: Gửi (push) tập tin từ Git lên Github với lệnh

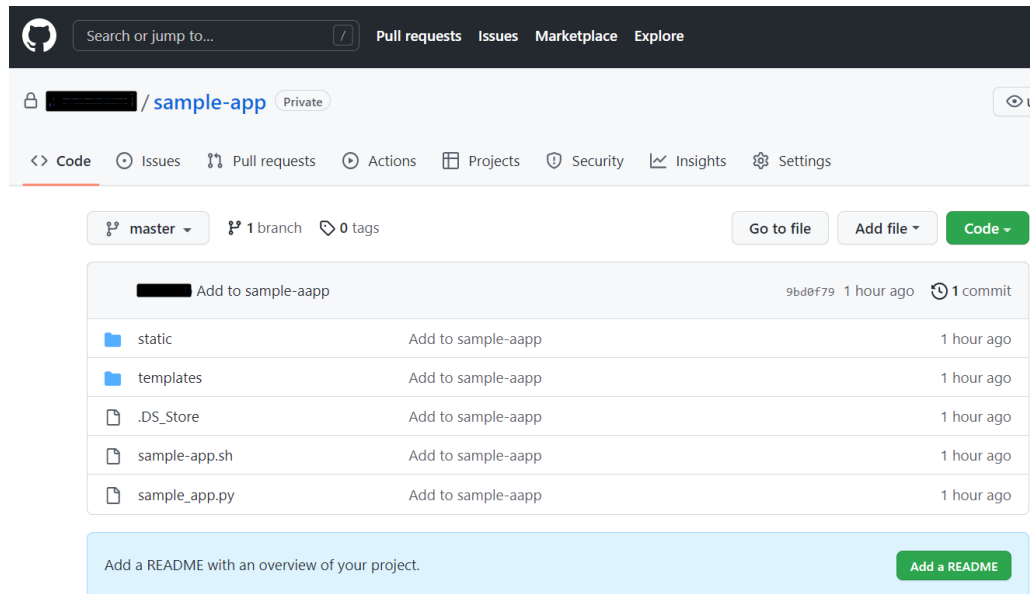
Gợi ý: dùng lệnh **git push origin master**, dùng token thay vì password.

- Bước 11: Kiểm tra tập tin trên Github

B.2 Xây CI/CD Pipeline bằng Jenkins

B.2.1 Commit Sample App lên Github

Sinh viên giải nén **sample-app.zip**, sau đó tạo GitHub repository để commit sample-app, với các bước tương tự ở **B.1.6**. Sau khi thực hiện thành công, ta thu được kết quả như bên dưới.



B.2.2 Sửa đổi Sample App và push thay đổi lên Git

Do mặc định Sample App và Jenkins Docker cùng dùng port 8080, nên ta thay đổi port trong mã nguồn của Sample App.

Bước 1. Mở tập tin **sample_app.py**, chỉnh sửa port 8080 thành 5050 ở dòng bên dưới:

```
if __name__ == "__main__":
    sample.run(host="0.0.0.0", port=5050)
```

Bước 2. Mở tập tin **sample-app.sh**, cũng chỉnh sửa port 8080 thành 5050:

```
...
echo "COPY sample_app.py /home/myapp/" >> tempdir/Dockerfile
echo "EXPOSE 5050" >> tempdir/Dockerfile
echo "CMD python /home/myapp/sample_app.py" >> tempdir/Dockerfile

cd tempdir
docker build -t sampleapp .
docker run -t -d -p 5050:5050 --name samplerunning sampleapp
docker ps -a
```

Bước 3. Build và kiểm tra sample-app

- Thực thi file **sample-app.sh** để chạy ứng dụng với port mới 5050

```
$ sudo bash ./sample-app.sh
```

- Mở 1 terminal khác hoặc trình duyệt, kiểm tra đường dẫn <http://localhost:5050> hoặc http://<ip_linux_vm>:5050

```
$ curl localhost:5050
```

Hoặc



Bước 4. Push sự thay đổi lên GitHub

Yêu cầu 3. Sinh viên thực hiện commit và push code với thông điệp “**Changed port from 8080 to 5050**”, trình bày step-by-step có minh chứng.

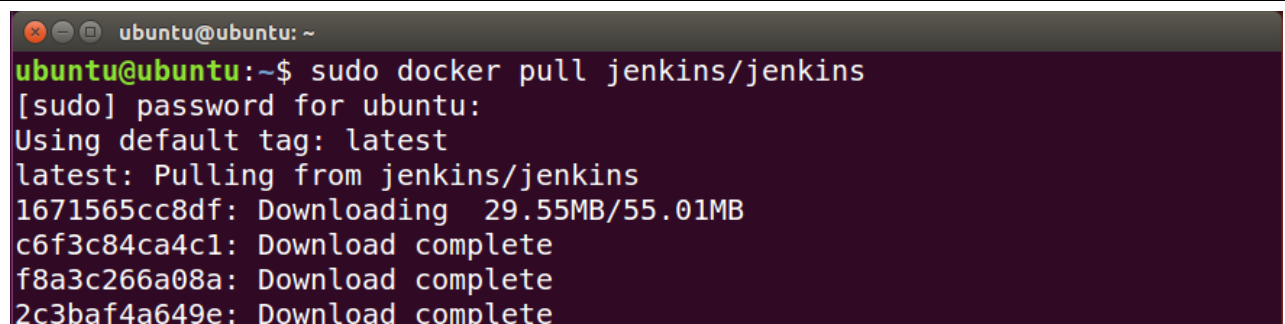
B.2.3 Tải và thiết lập chạy Jenkins Docker Image

Bước 1. Tải Jenkins Docker image

Jenkins Docker image được lưu trữ tại đây <https://hub.docker.com/r/jenkins/jenkins>

Tại thời điểm soạn bài lab này, lệnh để tải xuống Jenkins:

```
$ sudo docker pull jenkins/jenkins
```



Bước 2. Khởi chạy Jenkins Docker container

Lệnh sau sẽ khởi chạy Jenkins Docker container và sau đó cho phép các lệnh Docker được thực thi bên trong máy chủ Jenkins.

```
$ sudo docker run --rm -u root -p 8080:8080 -v jenkins-data:/var/jenkins_home -v $(which docker):/usr/bin/docker -v /var/run/docker.sock:/var/run/docker.sock -v "$HOME":/home --name jenkins_server jenkins/jenkins:lts
```

Các tùy chọn được sử dụng trong lệnh chạy docker này như sau:

- **--rm:** Tùy chọn này tự động xóa Docker container khi bạn ngừng chạy nó.

- **-u:** Tùy chọn này chỉ định người dùng. Ta muốn Docker container này chạy dưới dạng root để tất cả các lệnh Docker được nhập bên trong máy chủ Jenkins được cho phép thực thi
- **-p:** Tùy chọn này chỉ định port mà máy chủ Jenkins sẽ chạy cục bộ (local).

Yêu cầu 4. Sinh viên giải thích ý nghĩa của option -v trong lệnh chạy Jenkins?

Bước 3. Máy chủ Jenkins đã khởi chạy xong khi có dòng thông báo như bên dưới.

```
Jenkins initial setup is required. An admin user has been created and a
password generated.
Please use the following password to proceed to installation:

399b1d4a2c22443aa33a2bb17d669408

This may also be found at: /var/jenkins_home/secrets/initialAdminPassword

*****
*****
*****

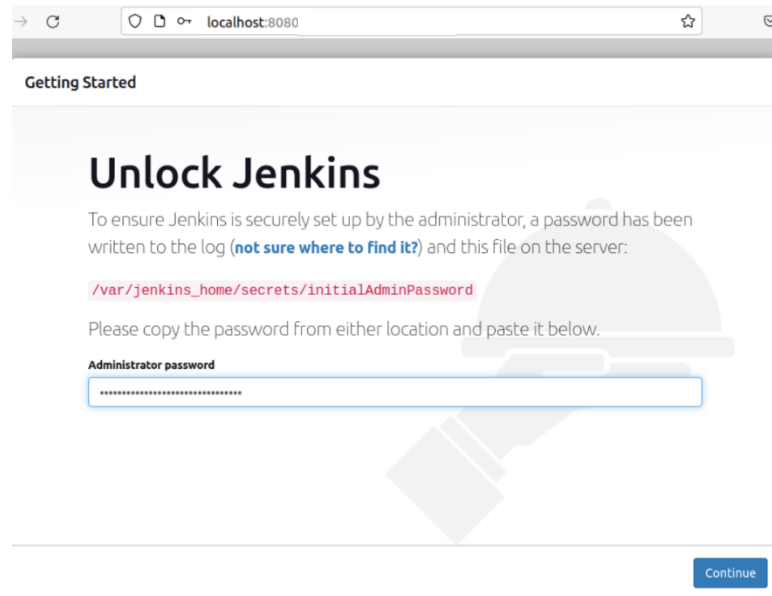
2022-09-11 16:57:13.300+0000 [id=28] INFO jenkins.InitReactorRunner$1#onAttained: Completed initialization
2022-09-11 16:57:13.383+0000 [id=22] INFO hudson.lifecycle.Lifecycle#onReady: Jenkins is fully up and running
```

Lưu ý bên trên có dòng password cần lưu lại để sử dụng về sau. Sau khi khởi chạy thành công, có thể hình dung hệ thống đang chạy theo biểu đồ dưới đây, để có cái nhìn về các mức độ hệ thống Docker-inside-Docker.

```
+-----+
|Your Computer's Operating System|
| +-----+ |
| |Linux VM| | | | |
| | +-----+ |
| | | Docker container | |
| | | +-----+ |
| | | | Jenkins server | |
| | | | +-----+ |
| | | | | Docker container | |
| | | | +-----+ |
| | | | +-----+ |
| | | +-----+ |
| | +-----+ |
| +-----+ |
+-----+
```

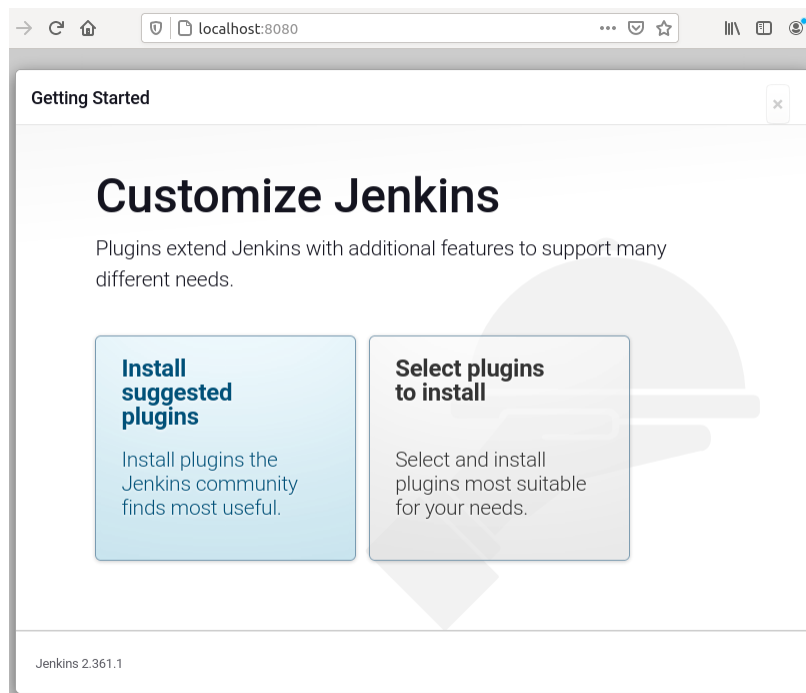

B.2.4 Cấu hình Jenkins

Bước 1. Mở trình duyệt và truy cập <http://localhost:8080/> hoặc <http://<ip linux vm>:8080/> và nhập mật khẩu admin đã lưu lại ở Bước trước.

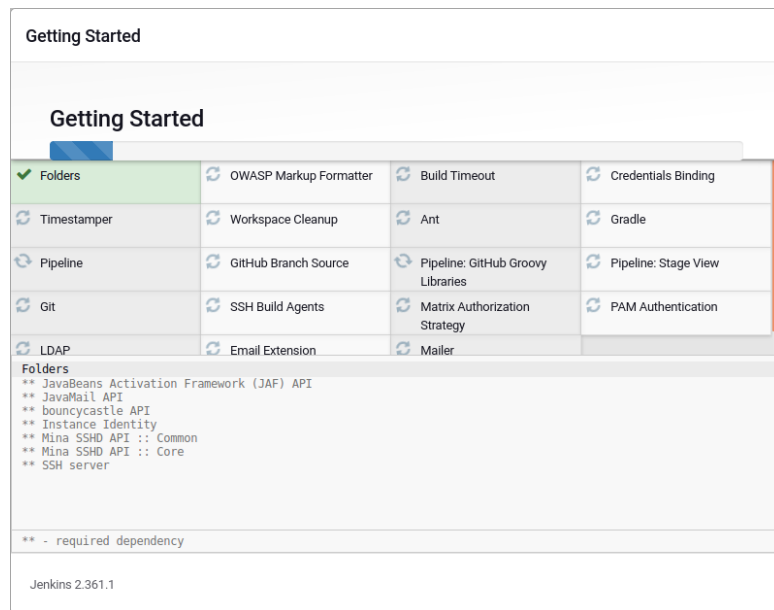


Bước 2. Cài đặt các Jenkins plugins khuyến nghị

Chọn vào Cài đặt các plugin được đề xuất và đợi Jenkins tải xuống và cài đặt các plugin.



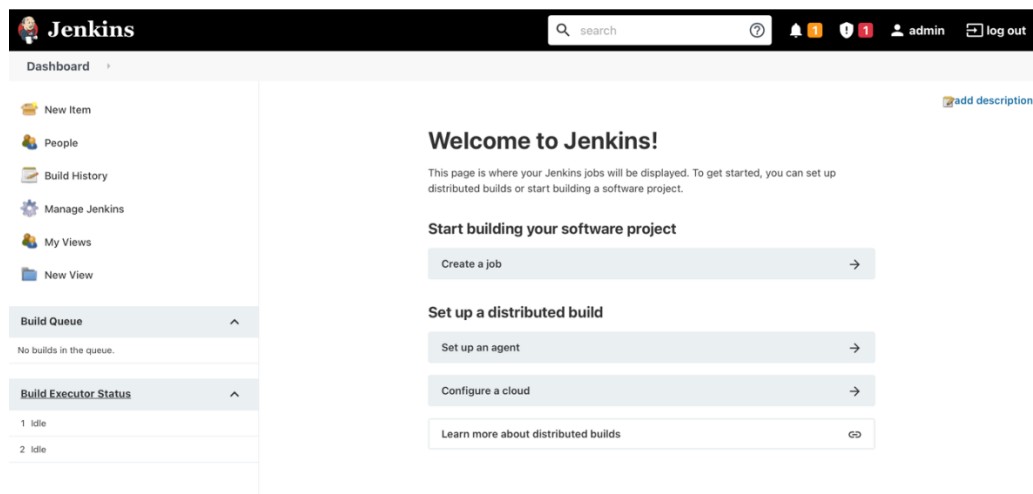
Trong cửa sổ terminal, ta sẽ thấy các thông báo log khi quá trình cài đặt được tiến hành. Giai đoạn này sẽ tốn khá nhiều thời gian.



Bước 3. Bỏ qua tạo tài khoản admin bằng cách chọn *Skip and continue as admin*.

Bước 4. Trong cửa sổ *Instance Configuration*, chọn tiếp *Save and Finish*.

Bước 5. Bắt đầu sử dụng Jenkins bằng cách chọn *Start using Jenkins*.



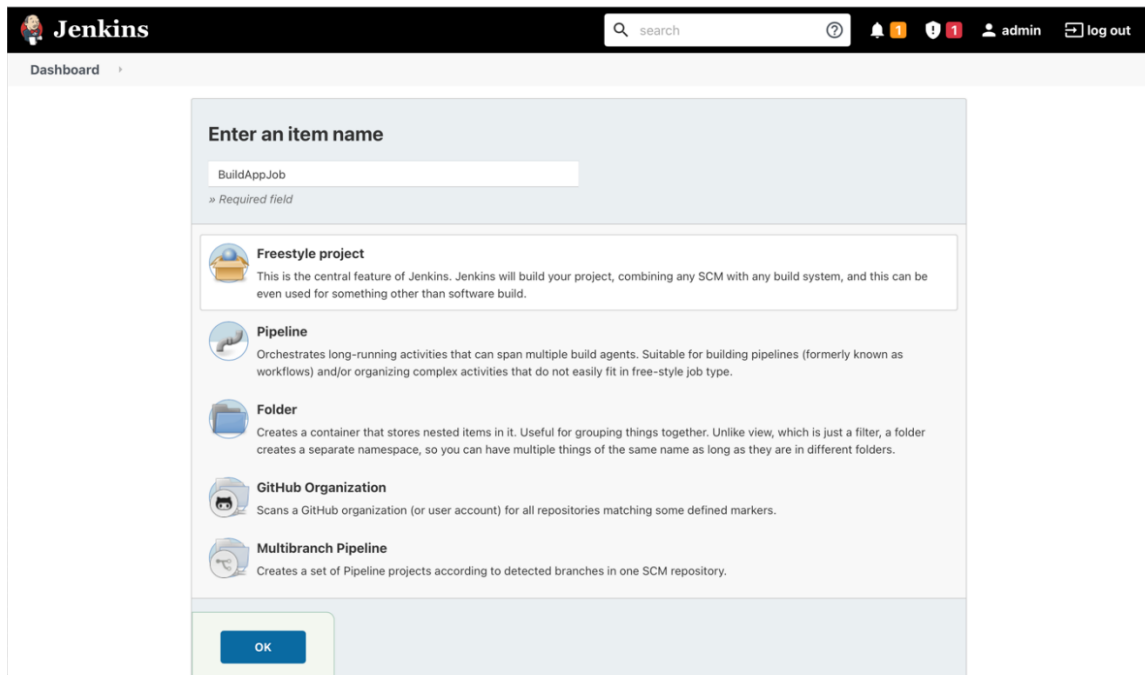
B.2.5 Sử dụng Jenkins để chạy ứng dụng đã dựng

Đơn vị cơ bản của Jenkins là job (hay project). Ta có thể tạo nhiều job với các tác vụ sau:

- Lấy mã nguồn từ repository Github
- Dựng lại ứng dụng và script hoặc build tool.
- Đóng gói ứng dụng và chạy nó trên một máy chủ.

Bước 1. Tạo job mới

- Chọn *Create a job*
- Trong trường *Enter an item name*, điền **BuildAppJob**
- Chọn thể loại *Freestyle project*
- Chọn *OK*

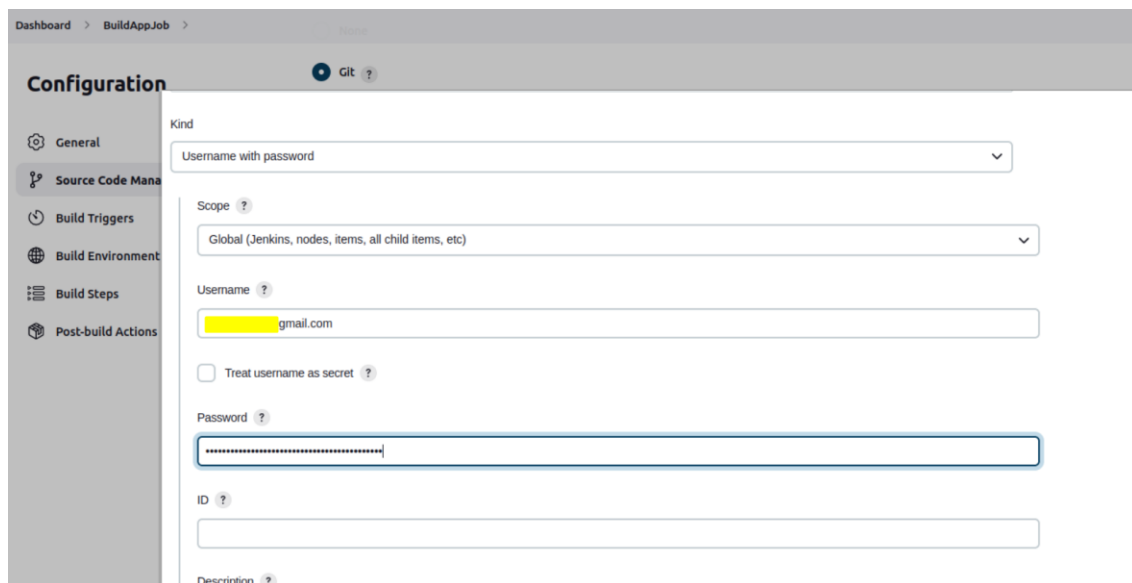


Bước 2. Cấu hình Jenkins BuildAppJob

Bây giờ ta đang ở cửa sổ cấu hình, nơi ta có thể nhập chi tiết cho job của mình. Các tab trên cùng là phím tắt cho các tùy chọn bên dưới.

- Chọn tab *General*, thêm mô tả cho job. Ví dụ: "**Nhom X's** first Jenkins job."
- Chọn tab *Source Code Management* và chọn *Git*. Thêm liên kết repository Github của ứng dụng. VD: <https://github.com/username/sample-app.git>
- Tại *Credentials*, chọn nút *Add* và chọn *Jenkins*
- Ở cửa sổ *Add Credentials*, điền GitHub username và password (token), chọn *Add*.

Lưu ý: Token cần được cấu hình gán đủ quyền truy cập để Jenkins sử dụng được.



- Quay lại dropdown *Credentials* tại tab *General*, chọn cấu hình credential vừa tạo.

Configuration

- General
- Source Code Management**
- Build Triggers
- Build Environment
- Build Steps
- Post-build Actions

Source Code Management

None

Git

Repositories

Repository URL:

Credentials:

[+ Add](#)

[Advanced...](#)

- Sau khi ta đã thêm đúng URL và thông tin đăng nhập, Jenkins kiểm tra quyền truy cập vào repository. Nếu có thông báo lỗi (màu đỏ) thì kiểm tra lại bước trên.
- Chọn tab *Build Steps*. Ở dropdown *Add build step*, chọn *Execute shell*/

Dashboard > BuildAppJob >

Configuration

- General
- Source Code Management
- Build Triggers
- Build Environment**
- Build Steps
- Post-build Actions

Build Environment

- ☐ Delete workspace before build starts
- ☐ Use secret text(s) or file(s)
- ☐ Add timestamps to the Console Output
- ☐ Inspect build log for published Gradle build scans
- ☐ Terminate a build if it's stuck
- ☐ With Ant

Build Steps

[Add build step](#)

Filter

- Execute Windows batch command
- Execute shell**
- Invoke Ant
- Invoke Gradle script

- Trong trường *Command*, nhập lệnh ta sử dụng để chạy ứng dụng *sample-app.sh*

Dashboard > BuildAppJob >

Configuration

- General
- Source Code Management
- Build Triggers
- Build Environment
- Build Steps**
- Post-build Actions

Build Steps

Execute shell

Command

See [the list of available environment variables](#)

`bash ./sample-app.sh`

[Advanced...](#)

- Chọn *Save*. Ta quay trở lại Jenkins dashboard với BuildAppJob đã được chọn.

Bước 3. Dùng Jenkins để build ứng dụng

Lưu ý: trước khi build cần xóa container samplerunning đang chạy app sample-app trên máy ảo để không bị đùng độ (dùng lệnh docker stop và docker rm).

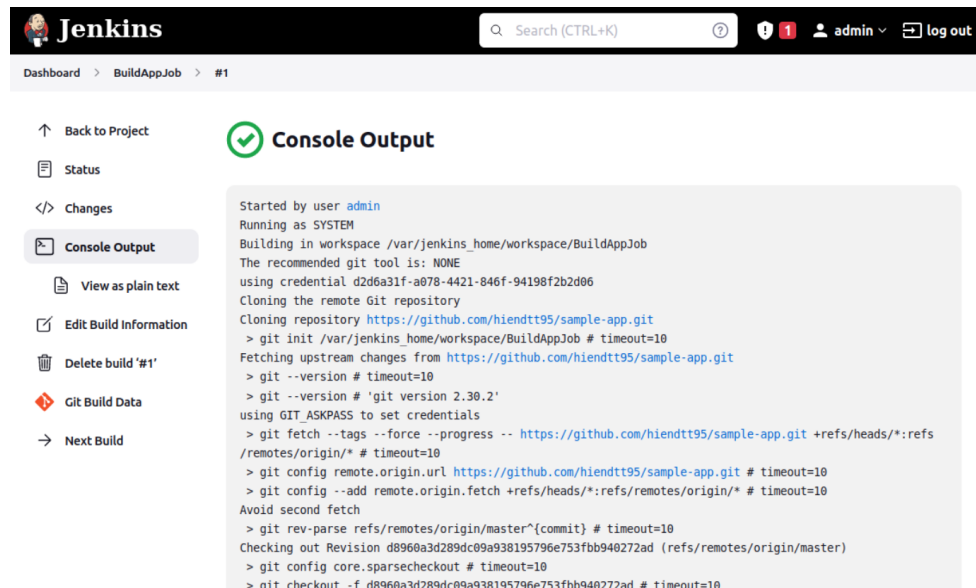
Ở menu bên trái, chọn **Build Now** để bắt đầu job. Jenkins sẽ tải xuống repository Git của chúng ta và thực thi lệnh chạy `bash ./sample-app.sh`.

Bước 4. Truy cập chi tiết bản dựng

Trong phần **Build History**, chọn 1 bản chạy ứng dụng bất kỳ với ký hiệu `#build number`.

Bước 5. Xem console output

Chọn **Console Output**.



```

Started by user admin
Running as SYSTEM
Building in workspace /var/jenkins_home/workspace/BuildAppJob
The recommended git tool is: NONE
using credential d2d6a31f-a078-4421-846f-94198f2b2d06
Cloning the remote Git repository
Cloning repository https://github.com/hiendtt95/sample-app.git
> git init /var/jenkins_home/workspace/BuildAppJob # timeout=10
Fetching upstream changes from https://github.com/hiendtt95/sample-app.git
> git --version # timeout=10
> git --version # 'git version 2.30.2'
using GIT_ASKPASS to set credentials
> git fetch --tags --force --progress -- https://github.com/hiendtt95/sample-app.git +refs/heads/*:refs/remotes/origin/* # timeout=10
> git config remote.origin.url https://github.com/hiendtt95/sample-app.git # timeout=10
> git config --add remote.origin.fetch +refs/heads/*:refs/remotes/origin/* # timeout=10
Avoid second fetch
> git rev-parse refs/remotes/origin/master^{commit} # timeout=10
Checking out Revision d8960a3d289dc09a938195796e753fbb940272ad (refs/remotes/origin/master)
> git config core.sparsecheckout # timeout=10
> git checkout -f d8960a3d289dc09a938195796e753fbb940272ad # timeout=10
  
```

Thông báo thành công và kết quả lệnh `docker ps -a`. Ở container, ứng dụng vừa dựng chạy ở port 5050 và Jenkins server là 8080.

```

---> 0c8b058e3e6f
Successfully built 0c8b058e3e6f
Successfully tagged sampleapp:latest
0b26a16e1d01747288c68c130b8458529c8a6a6ecc44916a231c4b28764c532de
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS
PORTS         NAMES
0b26a16e1d01   sampleapp     "/bin/sh -c 'python ..." Less than a second ago Up Less than a
second        0.0.0.0:5050->5050/tcp, :::5050->5050/tcp   samplerunning
097f6a77887b   jenkins/jenkins:lts "/usr/bin/tini -- /u..." 6 minutes ago Up 6 minutes
0.0.0.0:8080->8080/tcp, :::8080->8080/tcp, 50000/tcp   jenkins_server
Finished: SUCCESS
  
```

Bước 6. Mở trình duyệt và truy cập vào <https://localhost:5050> để xác nhận ứng dụng sampleapp đã chạy thành công.

B.2.6 Dùng Jenkins để kiểm tra bản dựng

Trong phần này ta sẽ tạo job thứ 2, kiểm tra build hoạt động bình thường.

Lưu ý: Cần đảm bảo đã dừng và xóa docker container samplerunning.

Yêu cầu 5. Sinh viên hoàn thành job kiểm tra bản dựng ứng dụng tự động theo gợi ý bên dưới, trình bày step-by-step có minh chứng.

Bước 1. Tạo job mới để kiểm tra **sample-app**

Yêu cầu đặt tên: *TestAppJob_NhomX*

Bước 2. Cấu hình Jenkins TestAppJob

- Đặt mô tả: "NhomX's first Jenkins test."
- *Source Code Management* chọn *None*
- Ở *Build Triggers* tab, chọn checkbox *Build after other projects are built* và điền *BuildAppJob* và *Trigger only if build is stable*

Bước 3. Viết tập lệnh thử nghiệm sẽ chạy sau khi dựng BuildAppJob

- Chọn *Build Steps*, chọn *Add build step* và chọn *Execute shell*.
- Yêu cầu viết bash shell:
 - Truy cập đường dẫn ứng dụng.
 - Nhận kết quả trả về.
 - Kiểm tra trong đó có tồn tại thông tin nhận biết thành công hay không. Nếu thành công thì exit code là 0 và ngược lại là 1.
- Chọn *Save*

Bước 4. Yêu cầu Jenkins chạy lại job BuildAppJob

Bước 5. Kiểm tra cả 2 job hoàn thành hay không

Nếu thành công, trong Dashboard ta sẽ thấy cột Last Success ở cả BuildAppJob và TestAppJob_NhomX.

B.2.7 Tạo Pipeline trong Jenkins

Mặc dù có thể chạy hai job của mình bằng cách chỉ cần nhấp vào nút **Build Now** cho BuildAppJob, nhưng các dự án phát triển phần mềm thường phức tạp hơn nhiều. Những dự án này có thể được hưởng lợi rất nhiều từ việc tự động hóa các bản dựng để tích hợp liên tục các thay đổi đoạn mã và liên tục tạo các bản dựng phát triển đã sẵn sàng để triển khai. Đây là bản chất của CI/CD. Một pipeline có thể được tự động hóa điều này.

Yêu cầu 6. Sinh viên hoàn thành pipeline của 2 ứng dụng Build và Test theo gợi ý bên dưới, trình bày step-by-step có minh chứng.

Bước 1. Tạo job Pipeline

- Chọn *New Item*
- Ở trường *Enter an item name*, điền ***SamplePipeline_NhomX***
- Chọn kiểu job Pipeline
- Chọn *OK*

Bước 2. Cấu hình job SamplePipeline_NhomX

- Ở tab *Pipeline*, trong phần *Script* điền đoạn mã sau:

```
node {
    stage('Preparation') {
        catchError(buildResult: 'SUCCESS') {
            sh 'docker stop samplerunning'
            sh 'docker rm samplerunning'
        }
    }
    stage('Build') {
        build 'BuildAppJob'
    }
    stage('Results') {
        build 'TestAppJob_NhomX'
    }
}
```

Sinh viên giải thích đoạn mã trên?

- Chọn *Save* và ta sẽ trở về Jenkins dashboard của SamplePipeline

Bước 3. Chạy SamplePipeline.

Chọn *Buid Now* để chạy job. Nếu code lỗi thì vào *Stage View* để xem log.

Bước 4. Kiểm tra SamplePipeline output

Ở mục *Build History*, chọn bản build với ký hiệu *#build number*, tiếp tục chọn *Console Output*.

B.3 Test Python Function với unittest

Sử dụng unittest để kiểm tra function có chức năng tìm kiếm đệ quy JSON object. Hàm trả về giá trị được gắn thẻ bằng một khoá cố định. Lập trình viên thông thường thực hiện hành động trên JSON object trả về bởi lời gọi API.

Bài test này sẽ sử dụng 3 tập tin như sau:

Tập tin	Chú thích
recursive_json_search.py	Đoạn mã này gồm function <code>json_search()</code> mà ta muốn test
test_data.py	Đây là dữ liệu mà hàm <code>json_search()</code> đang tìm kiếm
test_json_search.py	Đây là tập ta sẽ tạo để kiểm tra hàm <code>json_search()</code> trong tập mã <code>recursive_json_search.py</code> .

Bước 1. Xem tập tin test_data.py

Mở `unittest/test_data.py` và kiểm tra nội dung của nó. Dữ liệu JSON này là điển hình của dữ liệu trả về. Dữ liệu mẫu đủ phức tạp để trở thành một bài test tốt. Ví dụ, nó có các loại dict và list xen kẽ.

```
GNU nano 4.8                                test_data.py
key1 = "issueSummary"
key2 = "XY&^$#@!1234%^&"

data = {
    "id": "AWcvsjx864kVeDHDi2gB",
    "instanceId": "E-NETWORK-EVENT-AWcvsjx864kVeDHDi2gB-1542693469197",
    "category": "Warn",
    "status": "NEW",
    "timestamp": 1542693469197,
    "severity": "P1",
    "domain": "Availability",
    "source": "DNAC",
    "priority": "P1",
    "type": "Network",
    "title": "Device unreachable",
    "description": "This network device leaf2.abc.inc is unreachable from controller.",
    "actualServiceId": "10.10.20.82",
    "assignedTo": "",
    "enrichmentInfo": {
        "issueDetails": {
```

Bước 2. Tạo hàm `json_search()` mà ta sẽ kiểm tra.

Mở tập tin `unittest/recursive_json_search.py` và thêm hàm bên dưới.

```
from test_data import *
def json_search(key,input_object):
    ret_val=[]
    if isinstance(input_object, dict): # Iterate dictionary
        for k, v in input_object.items(): # searching key in the dict
            if k == key:
                temp={k:v}
                ret_val.append(temp)
            if isinstance(v, dict): # the value is another dict so repeat
                json_search(key,v)
            elif isinstance(v, list): # it's a list
                for item in v:
                    if not isinstance(item, (str,int)): # if dict or list
                        repeat
                            json_search(key,item)
            else: # Iterate a list because some APIs return JSON object in a list
                for val in input_object:
                    if not isinstance(val, (str,int)):
                        json_search(key,val)
    return ret_val
print(json_search("issueSummary",data))
```

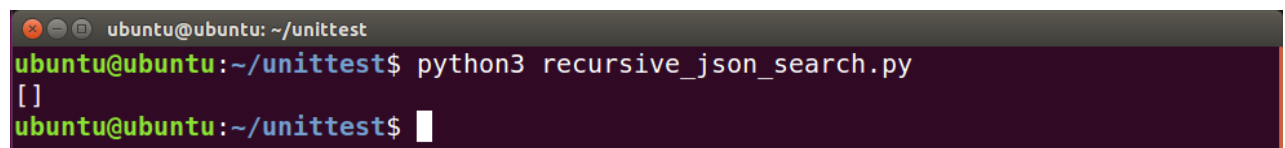
Hàm `json_search` nhận một khoá (key) và một JSON object làm tham số đầu vào, sau đó trả về list các cặp key/value. Phiên bản hiện tại của function cần kiểm tra để xem nó có hoạt động đúng như dự định hay không.

Hoạt động của function này:

- Trước tiên nhận dữ liệu test.
- Tìm kiếm dữ liệu phù hợp các biến key trong tập tin `test_data.py`. Nếu tìm ra một kết quả phù hợp, nó sẽ nối dữ liệu phù hợp vào list.
- Hàm `print()` ở cuối dùng để in list kết quả cho ví dụ `key = "issueSummary"`

Bước 3. Lưu và chạy code trên.

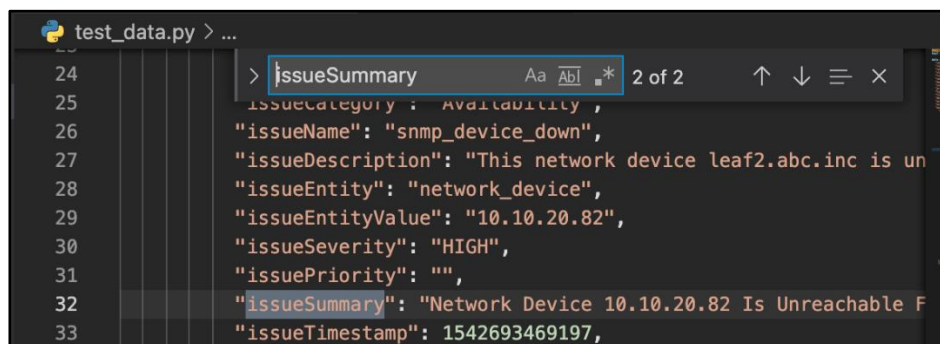
```
$ python3 recursive_json_search.py
```



```
ubuntu@ubuntu: ~/unittest
ubuntu@ubuntu:~/unittest$ python3 recursive_json_search.py
[]
ubuntu@ubuntu:~/unittest$
```

Kết quả chạy hàm `json_search()` cho thấy, key "issueSummary" không có trong dữ liệu JSON được trả về từ việc gọi API.

Bước 4. Kiểm tra lại kết quả của hàm `json_search()` bằng mở tập tin `test_data.py` và tìm kiếm "issueSummary".



```
test_data.py > ...
24 > |issueSummary      Aa Ab|_* 2 of 2  ↑ ↓ ≡ ×
25 |issuecategory : Availability ,
26 |"issueName": "snmp_device_down",
27 |"issueDescription": "This network device leaf2.abc.inc is un
28 |"issueEntity": "network_device",
29 |"issueEntityValue": "10.10.20.82",
30 |"issueSeverity": "HIGH",
31 |"issuePriority": "",
32 |"issueSummary": "Network Device 10.10.20.82 Is Unreachable F
33 |"issueTimestamp": 1542693469197,
```

⇒ Đoạn code lỗi.

Bước 5. Tạo một unit test để kiểm tra function có hoạt động đúng như dự kiến

- Mở tập tin `test_json_search.py`
- Dòng đầu tiên thêm thư viện `unittest`

```
import unittest
```

- Thêm các dòng import function đang testing cũng như dữ liệu JSON mà hàm sử dụng

```
from recursive_json_search import *
from test_data import *
```

- Thêm đoạn mã vào class `json_search_test`. Mã tạo ra subclass `TestCase` của `unittest` framework. Class định nghĩa một số phương pháp kiểm tra được sử dụng trên function `json_search()` trong `recursive_json_search.py`. Lưu ý mỗi phương thức test bắt đầu với **`test_`**, cho phép `unittest` framework tự động khám phá. Thêm các dòng sau vào cuối `test_json_search.py`

```
class json_search_test(unittest.TestCase):
    '''test module to test search function in
    `recursive_json_search.py`'''
    def test_search_found(self):
        '''key should be found, return list should not be empty'''
        self.assertTrue([]!=json_search(key1,data))
    def test_search_not_found(self):
        '''key should not be found, should return an empty list'''
        self.assertTrue([]==json_search(key2,data))
    def test_is_a_list(self):
        '''Should return a list'''
        self.assertIsInstance(json_search(key1,data),list)
```

- Trong code `unittest`, đang sử dụng ba phương pháp để test function tìm kiếm.
 1. Đưa ra một key tồn tại sẵn trong JSON object, xem đoạn code có thể tìm thấy key như vậy không.
 2. Đưa một key không tồn tại trong JSON object, xem liệu code có xác nhận rằng không có key nào tìm được.
 3. Kiểm tra xem function có trả về một list như mong đợi hay không.

Để tạo các test, đoạn code sử dụng phương thức `assert` được tích hợp trong class `unittest TestCase` để kiểm tra các điều kiện. Phương thức `assertTrue(x)` sẽ kiểm tra một điều kiện có đúng không, `assertIsInstance(a,b)` kiểm tra xem `a` có phải là một thể hiện của kiểu `b` hay không. Thể hiện kiểu ở đây là list.

Lưu ý mỗi phương thức đều có chú thích trong nháy đôi ("`''`"), điều này bắt buộc nếu kiểm tra thông tin output của phương pháp test khi chạy.

- Cuối tập tin, thêm phương thức `unittest.main()`, điều này cho phép `unittest` chạy từ command line

```
if __name__ == '__main__':
    unittest.main()
```

Bước 6. Chạy test để xem kết quả

- Chạy code test.

- Quan sát kết quả: Đầu tiên ta thấy list trống, thứ hai ta thấy **F.** Dấu chấm có nghĩa test passed và điểm F có nghĩa là test failed. Do đó, phép thử thứ nhất đã passed, phép thử thứ hai test failed, và phép thử thứ ba cũng test passed.

```
ubuntu@ubuntu: ~/unittest
ubuntu@ubuntu:~/unittest$ python3 test_json_search.py
[]
.F.

=====
FAIL: test_search_found (__main__.json_search_test)
key should be found, return list should not be empty
=====
Traceback (most recent call last):
  File "test_json_search.py", line 9, in test_search_found
    self.assertTrue([]!=json_search(key1,data))
AssertionError: False is not true
=====
Ran 3 tests in 0.001s

FAILED (failures=1)
```

- Để liệt kê từng bài test và kết quả của nó, thêm tùy chọn -v trong lệnh unittest.

```
ubuntu@ubuntu: ~/unittest
ubuntu@ubuntu:~/unittest$ python3 -m unittest -v test_json_search.py
[]
test_is_a_list (test_json_search.json_search_test)
Should return a list ... ok
test_search_found (test_json_search.json_search_test)
key should be found, return list should not be empty ... FAIL
test_search_not_found (test_json_search.json_search_test)
key should not be found, should return an empty list ... ok

=====
FAIL: test_search_found (test_json_search.json_search_test)
key should be found, return list should not be empty
=====
Traceback (most recent call last):
  File "/home/ubuntu/unittest/test_json_search.py", line 9, in test_search_found
    self.assertTrue([]!=json_search(key1,data))
AssertionError: False is not true
=====
Ran 3 tests in 0.003s

FAILED (failures=1)
```

Bước 7. Điều tra và sửa lỗi trong đoạn mã recursive_json_search.py

key should be found, return list should not be empty ... FAIL => Khoá không tìm được. Tại sao? Ta nhìn vào đoạn code của hàm đệ quy thấy rằng `ret_val=[]` đang được thực thi lặp đi lặp lại mỗi lần hàm được gọi. Điều này làm cho list luôn rỗng và làm mất kết quả tích lũy từ lệnh `ret_val.append(temp)` đang thêm vào danh sách `ret_val`.

```
def json_search(key,input_object):
    ret_val=[]
    if isinstance(input_object, dict): # Iterate dictionary
        for k, v in input_object.items(): # searching key in the dict
            if k == key:
                temp={k:v}
                ret_val.append(temp)
```

Yêu cầu 7. Sinh viên sửa lại lỗi theo gợi ý trên và thực thi lại unittest. Nếu còn lỗi, tiếp tục chỉnh sửa đến khi pass cả 3 phép thử; trình bày step-by-step có minh chứng.

Gợi ý:

- Dòng `ret_val=[]` bị ghi đè giá trị rỗng mỗi lần gọi đệ quy hàm, gây ra lỗi, cần di chuyển `ret_val` ra bên ngoài hàm `json_search`.
- Lưu ý khi di chuyển `ret_val`, nếu biến này trở thành 1 biến toàn cục (global variable), trong 1 lần chạy file, gọi hàm `json_search` nhiều lần sẽ khiến lần gọi sau sử dụng lại kết quả `ret_val` của lần gọi trước. Vậy cần sửa như thế nào?

C YÊU CẦU & ĐÁNH GIÁ

- Sinh viên tìm hiểu và thực hành theo hướng dẫn.
- Sinh viên báo cáo kết quả thực hiện và nộp bài bằng **1 trong 2 hình thức**:

C.1 Cách 1: Báo cáo trực tiếp trên lớp

Báo cáo trực tiếp kết quả thực hành (có hình ảnh minh họa các bước) với GVTH trong buổi học, trả lời các câu hỏi và giải thích các vấn đề kèm theo.

C.2 Cách 2: Nộp file báo cáo

Báo cáo cụ thể quá trình thực hành (có hình ảnh minh họa các bước), trả lời các câu hỏi và giải thích các vấn đề kèm theo trong file PDF theo mẫu tại website môn học.

Đặt tên file báo cáo theo định dạng như mẫu:

[Mã lớp]-LabX_MSSV1-Tên SV1_MSSV2 -Tên SV2

Ví dụ: *[NT521.N11.1]-Lab1_14520000-Viet_14520999-Nam.*

- Nếu báo cáo có nhiều file, nén tất cả file vào file .ZIP với cùng tên file báo cáo.
- Nộp báo cáo trên theo thời gian đã thống nhất tại website môn học.

D ĐÁNH GIÁ

- Sinh viên hiểu và tự thực hiện được bài thực hành, đóng góp tích cực tại lớp.
- Báo cáo trình bày chi tiết, giải thích các bước thực hiện và chứng minh được do nhóm sinh viên thực hiện.
- Hoàn tất nội dung cơ bản và thực hiện nội dung *mở rộng*

Kết quả thực hành cũng được đánh giá bằng kiểm tra kết quả trực tiếp tại lớp vào cuối buổi thực hành hoặc vào buổi thực hành thứ 2.

Lưu ý: Bài sao chép, nộp trễ, “*gánh team*”, ... sẽ được xử lý tùy mức độ.

HẾT

Chúc các bạn hoàn thành tốt!