Part 2 – Continuous Integration Setup

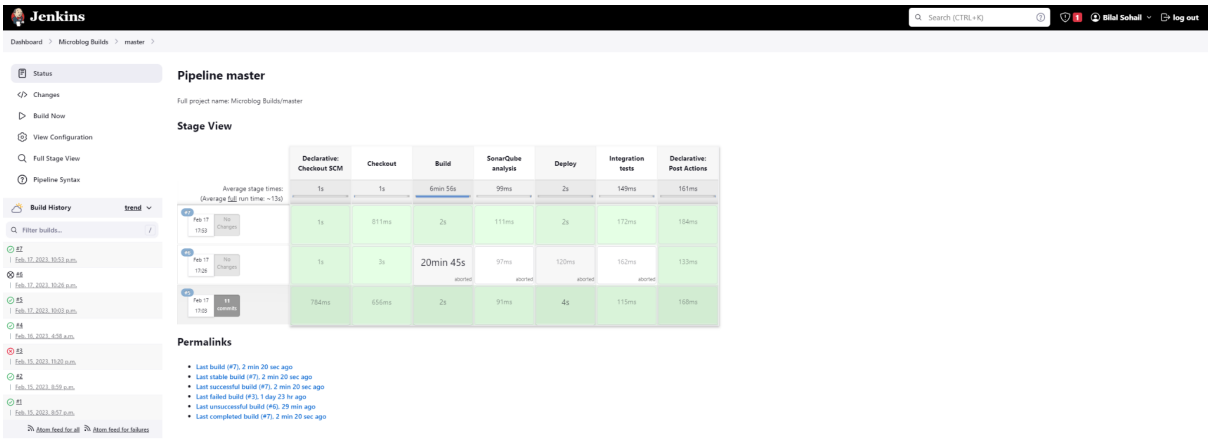
Written by: Bilal, Edxio, Nigel, Nitin, & Sophie

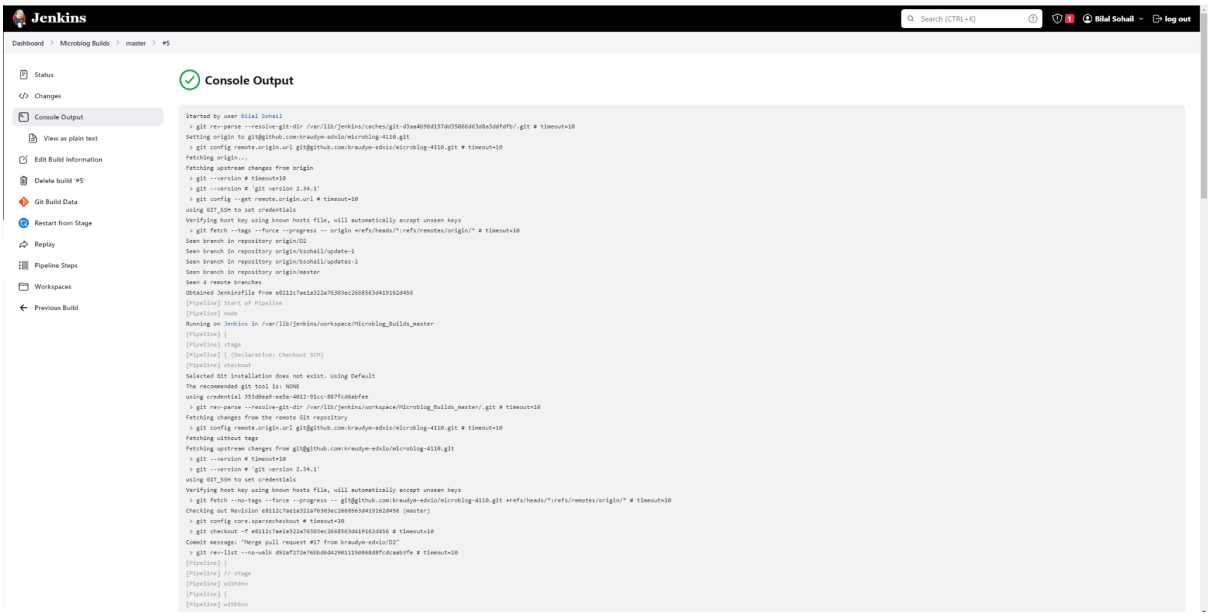
Date: Friday, February 17, 2023

i) Step-by-step demonstration of the Docker container being deployed with screenshots

The Step by step demonstration of the Docker container being built and deployed can be seen in the screenshots below:

(The build can be manually triggered by clicking on the '*Build Now*' button.)





Dashboard > Microblog Builds > master > #5

```
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Checkout)
[Pipeline] checkout
Selected Git installation does not exist. Using Default
The recommended git tool is: NONE
using credential 353d0ea9-ea5a-4012-91cc-887fcd6abfee
 > git rev-parse --resolve-git-dir /var/lib/jenkins/workspace/Microblog_Builds_master/.git # timeout=10
Fetching changes from the remote Git repository
 > git config remote.origin.url git@github.com:kraudym-edxio/microblog-4110.git # timeout=10
Fetching upstream changes from git@github.com:kraudym-edxio/microblog-4110.git
 > git --version # timeout=10
 > git --version # 'git version 2.34.1'
using GIT_SSH to set credentials
Verifying host key using known hosts file, will automatically accept unseen keys
 > git fetch --tags --force --progress -- git@github.com:kraudym-edxio/microblog-4110.git +refs/heads/*:refs/remotes/origin/* # timeout=10
 > git rev-parse refs/remotes/origin/master^{commit} # timeout=10
Checking out Revision e8112c7ae1a322a76303ec2668563d419162d456 (refs/remotes/origin/master)
 > git config core.sparsecheckout # timeout=10
 > git checkout -f e8112c7ae1a322a76303ec2668563d419162d456 # timeout=10
Commit message: "Merge pull request #17 from kraudym-edxio/D2"
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Build)
[Pipeline] echo
Building
[Pipeline] sh
+ sudo docker build --tag flaskapp .
Sending build context to Docker daemon  422.9kB

Step 1/16 : FROM python:slim
 ---> 281d606e8b48
Step 2/16 : RUN useradd microblog
 ---> Using cache
 ---> e2ef6d20eb24
Step 3/16 : WORKDIR /home/microblog
 ---> Using cache
 ---> c14388b2da23
Step 4/16 : COPY requirements.txt requirements.txt
 ---> Using cache
 ---> 5d8226f92228
Step 5/16 : RUN python -m venv venv
 ---> Using cache
 ---> 30d480a86862
Step 6/16 : RUN venv/bin/pip install -r requirements.txt
 ---> Using cache
 ---> d078ddf070ff
```

Dashboard > Microblog Builds > master > #5

```
 ---> f4ef0a2fff85
Step 10/16 : COPY microblog.py config.py boot.sh ./
 ---> Using cache
 ---> ffbb3872ec77
Step 11/16 : RUN chmod a+x boot.sh
 ---> Using cache
 ---> 53afded7ade3
Step 12/16 : ENV FLASK_APP microblog.py
 ---> Using cache
 ---> 2523f65e31b3
Step 13/16 : RUN chown -R microblog:microblog ./
 ---> Using cache
 ---> f4749b34fc25
Step 14/16 : USER microblog
 ---> Using cache
 ---> 21755376bdaa
Step 15/16 : EXPOSE 5000
 ---> Using cache
 ---> 79cae760f9a5
Step 16/16 : ENTRYPOINT ["./boot.sh"]
 ---> Using cache
 ---> a3f9e1e3983d
Successfully built a3f9e1e3983d
Successfully tagged flaskapp:latest
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (SonarQube analysis)
[Pipeline] echo
SonarQube analysis
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Deploy)
[Pipeline] echo
Deploying
[Pipeline] sh
+ sudo docker stop microblogApp1
microblogApp1
[Pipeline] sh
+ sudo docker rm microblogApp1
microblogApp1
[Pipeline] sh
+ sudo docker run -d -p 5000:5000 --name microblogApp1 flaskapp
b029eb271ce9313affde318839f5ee465b3713d93daee309b6ad3bb0ebec5ef0
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Integration tests)
[Pipeline] echo
Integration tests
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Declarative: Post Actions)
[Pipeline] echo
Finished
```

Dashboard > Microblog Builds > master > #5

```
 ---> 79cae760f9a5
Step 16/16 : ENTRYPOINT ["./boot.sh"]
 ---> Using cache
 ---> a3f9e1e3983d
Successfully built a3f9e1e3983d
Successfully tagged flaskapp:latest
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (SonarQube analysis)
[Pipeline] echo
SonarQube analysis
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Deploy)
[Pipeline] echo
Deploying
[Pipeline] sh
+ sudo docker stop microblogApp1
microblogApp1
[Pipeline] sh
+ sudo docker rm microblogApp1
microblogApp1
[Pipeline] sh
+ sudo docker run -d -p 5000:5000 --name microblogApp1 flaskapp
b029eb271ce9313affde318839f5ee465b3713d93daee309b6ad3bb0ebec5ef0
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Integration tests)
[Pipeline] echo
Integration tests
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Declarative: Post Actions)
[Pipeline] echo
Finished
[Pipeline] echo
Pipeline succeeded
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

ii) An explanation of:

(1) What are Docker Containers and Images

Docker containers are executable components that allow the source code to be run on different environments. A docker image is an executable package that contains all of the requirements necessary to run the application. The difference between docker containers and docker images is that the images are read-only files, while the containers can be modified and interacted with.

(2) What is the difference of purpose between the Docker and the Docker-compose tools

Docker is used to manage single container applications. Docker compose is used on multi-container applications, and allows the user to allow other users to contribute.

(3) How is the Microblog app image created, included which tools are involved in the process

The Microblog app image is created in the Jenkinsfile after running the build in the multibranch pipeline. The tools involved in the process are Jenkins, docker, and flask. First an environment is configured with the name of the container, the image, the job along with the build url. During the build stage of the pipeline an image is created using the Dockerfile. Then in the deploy stage the container is run to deploy the application on the build url specified in the environment.

iii) Step-by-step description of the Jenkins and the Github setup with screenshots and an explanation for each configuration choice and action taken

Preliminary Steps:

- Setting up AWS
- Downloading the WS EC2 Node SSH Access Key
- SSH into the AWS ubuntu VM using the SSH Access Key
- Installing Jenkins on the Ubuntu VM as follows:

```
sudo apt install openjdk -11 - jdk -y
```

```
curl - fsSL https :// pkg . jenkins . io / debian - stable / jenkins . io . key
```

```
| sudo tee / usr / share / keyrings / jenkins - keyring . asc & gt ; / dev / null
```

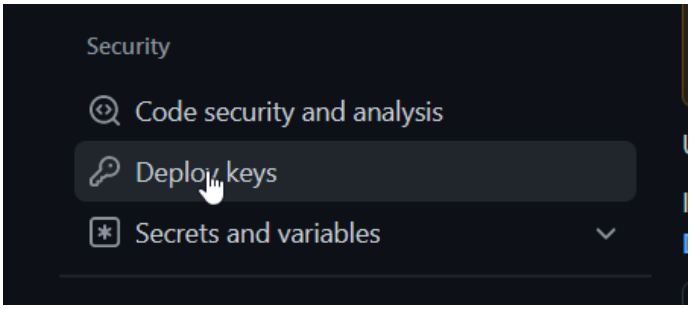
```
sudo apt install jenkins -y
```

Secondary Steps:

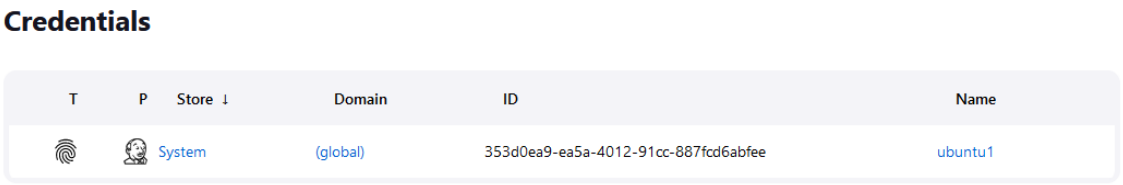
- Run Jenkins on the VM:

```
sudo systemctl enable -- now jenkins
```

- Add a deployment key to GitHub repository

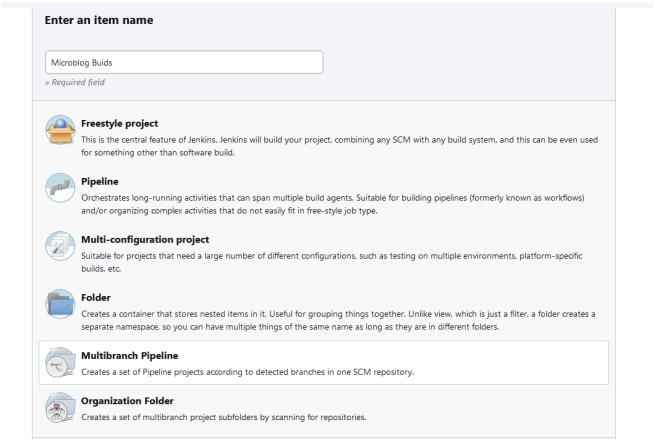


- Add the SSH key's credentials to the Jenkins Credential section



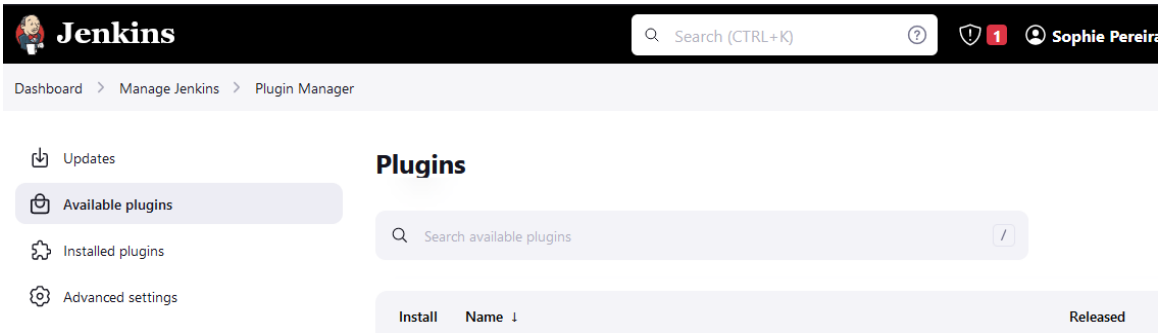
- Change setting in Manage Jenkins -> Configure Global security -> Git Host Key Verification Configuration and change the Host Key Verification Strategy to Accept first connection

- Create the Multibranch Pipeline by clicking New Item on the dashboard



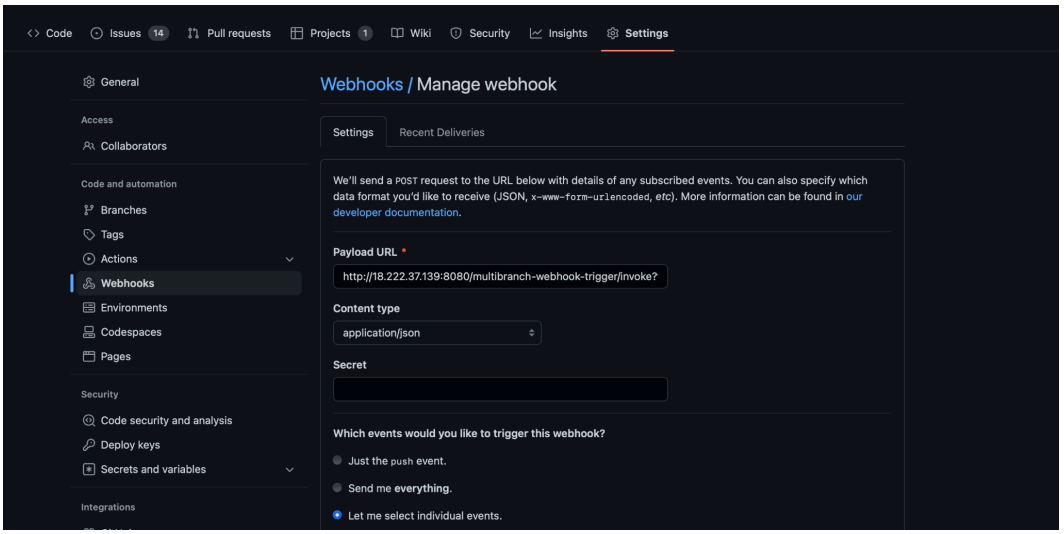
Extensions & Extras:

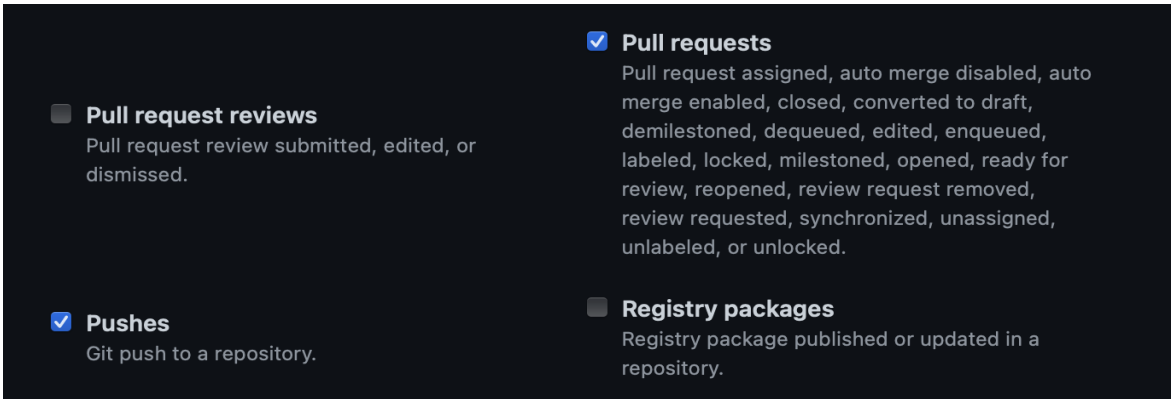
- Plugins can be added by heading to Manage Jenkins -> Plugin Manager and then searching for the plugin name in Available plugins



- Added SonarQube plugin to prepare setting up SonarQube scan in future deliverables
- Added Discord Notifier plugin to send a notification to a discord channel when there is a failure

- Added Webhook between Jenkins and GitHub to notify the Jenkins server of every push





- Added [Multibranch Scan Webhook Trigger] plugin to make the webhook possible in the multibranch pipeline

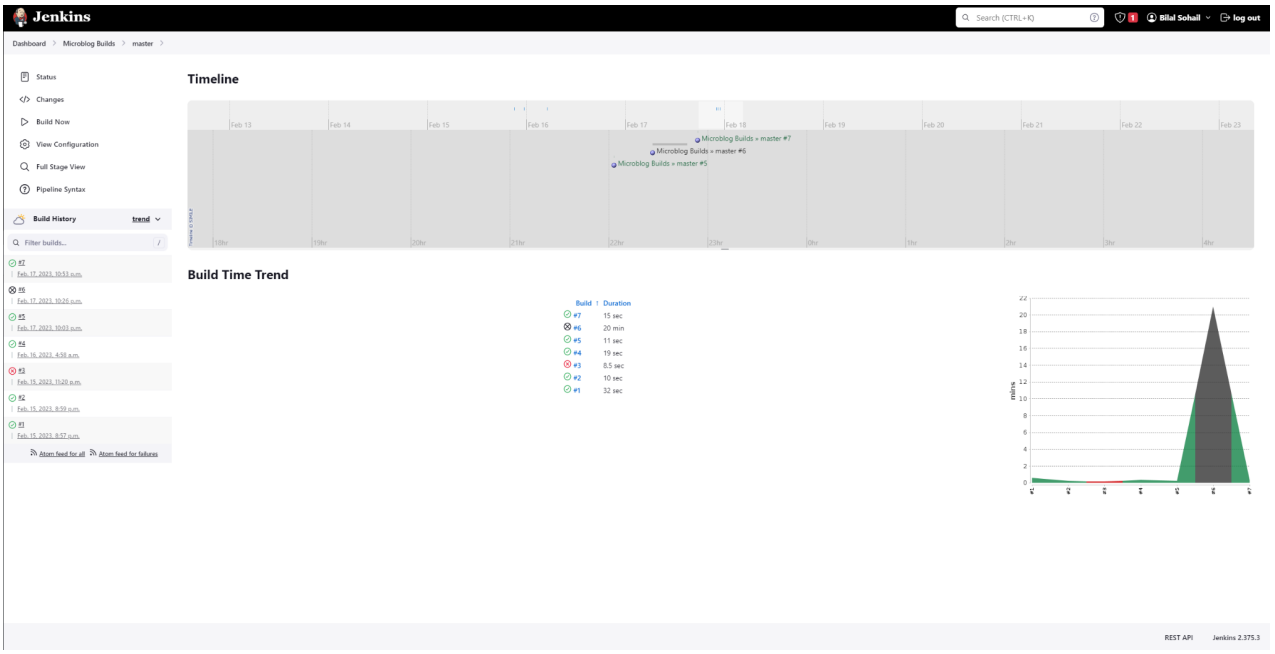
- Admin user created Jenkins account for all other group members, under Manage Jenkins -> Configure and add new users and click on the Create User button

iv) Description of every change made to the application files in order to support the CI process, including the Jenkinsfile and the Docker files

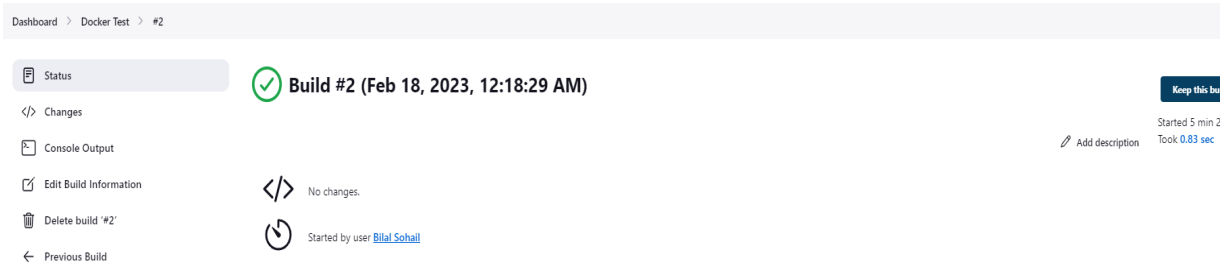
- Dockerfile was not modified
- Changed in requirements.txt the greenlet version to version 2.0.2
- Created the Jenkinsfile with a stage for Checkout, Build, SonarQube analysis, Deploy, and Integration tests along with a mechanism that finishes the pipeline with the correct result, success when it succeeds, and fails when it fails. When the build fails a notification will be sent to a discord channel using a webhook.

v) Screenshots showing the build history, trends and other Jenkins reports

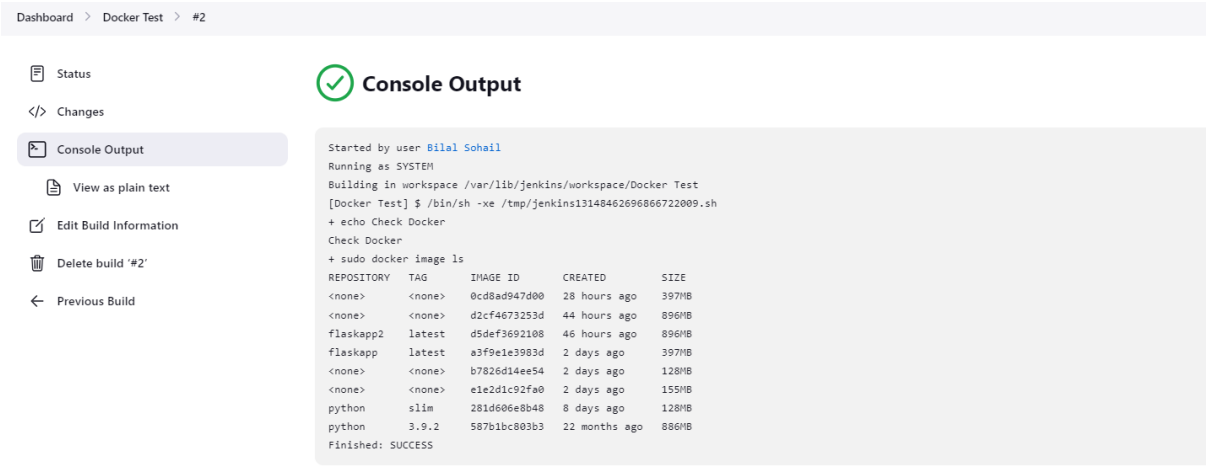
Build History and trends



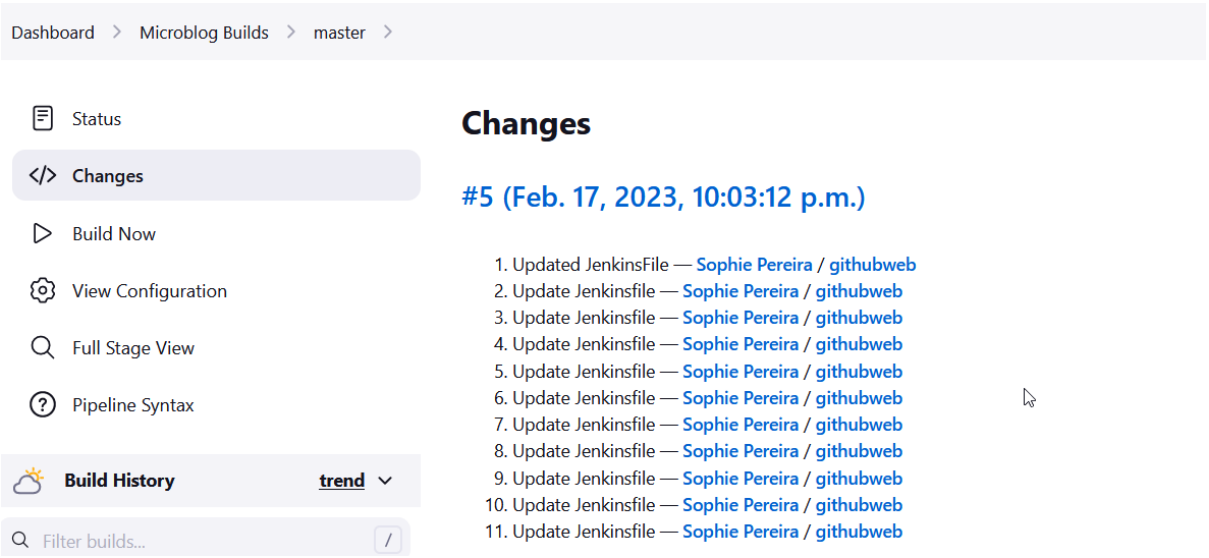
Build Status



Build Console Output



Changes

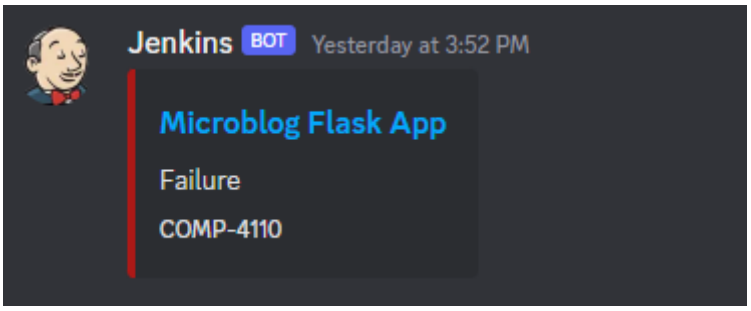


vi) An example of a notification when a build, scan, deployment, or test fails

When a build, scan, deployment, or test fails a notification will be sent to a discord channel using the [Discord Notifier plugin] and a webhook. This is how the plugin is used in the pipeline script:

```
post {
    always {
      echo 'Finished'
    }
  success {
      echo 'Pipeline succeeded'
  }
  failure {
      echo 'Pipeline failed'
      discordSend description: "Failure", footer: "COMP-4110", link: env.BUILD_URL, result: currentBuild.currentResult, title: JOB_NAME, webhookURL: "https://ptb.discord.com
```

Example of a notification being sent when a build, scan, deployment, or test fails:



Note: This is just an example of a notification when a pipeline fails. In the future, these notifications will be more detailed.

vii) A commentary on the utility of Continuous Integration for software development

Continuous integration (CI) is essentially the practice of integrating the team member's code changes into a shared controlled repository. Then, established automated tools are used to build, package and test the member's applications in order to confirm that software code is valid and error-free (or not) before it is integrated.

One of the key benefits of CI is that it helps catch bugs and integration issues early on in the development process. By automatically building, testing, and integrating code changes into a shared repository multiple times a day, developers can quickly identify and fix problems, which saves time and reduces the risk of delays. Another advantage of CI is that it makes it easier for team members to collaborate and share code changes. With CI, team members can be confident that their code changes will not break the build or cause problems for other members of the team. This results in a more streamlined and efficient development process, as well as better overall code quality.

In addition to improving software quality and collaboration, CI also helps increase the speed of delivery and reduces the time it takes to get new features and updates to users. By automating the build and test process, team members can focus on writing code, rather than spending time on manual testing.

Here's how CI packages test software builds:

(a) Developers enter their code into the shared repository.
(b) A CI server monitors the repository for new code changes(checks every few seconds).
(c) When new code is entered, the CI server automatically builds the code and runs a set of automated tests.
(d) If the test passes, the code is ready to be deployed.
(e) If the test fails, the developer is notified.

viii) Small proposal on how the Continuous Integration structure could be improved or augmented in this scenario

Jenkins is better suited for medium to large teams that run on a server as an open source tool with slow installation and configuration times. In this scenario the CI structure can be improved by using up-to-date and user friendly plugins. Jenkins plug-ins are coded by third-parties so the quality is at times inconsistent. Server administrator experience would be useful because Jenkins runs on a server. Setting changes need to be minimised because when the continuous integration pipeline breaks, the developer has to intervene.

In this scenario we could add additional stages to the multibranch pipeline. For example a stage for unit testing. Additional tools could also be added to improve the development process.

ix) Commentary on the Github flow model and a comparison with other commonly used branching models

The GitHub Flow is an ideal branching strategy for small teams because:

- Start with the main branch, then developers create branches
- Each developer works in his/her own space to isolate the code
- Then they can merge back to the master branch when the work is completed.
- The feature branch is then deleted.
- The branches protect the mainline of code and the changes within that branch do not affect other developers.
- The best advantage is that it enables parallel development.

As a result, the master code is in a constant deployable state and can support continuous integration and delivery processes.

Pros and Cons of GitHub Flow are:

- Allows for fast feedback to quickly identify issues and resolve them.
- No development branch - as you are testing and automating changes to one branch which allows for quick and continuous deployment.
- Ideal to maintain a single production version.
- Not suitable for handling multiple versions of the code.
- Leads to an unstable production code if branches are not properly tested before merging with the master-release preparation.

Pros and Cons of GitLab Flow:

- Combines feature-driven development and feature branching with issue tracking.
- Create a develop branch that defaults with the main branch right away.
- Great when you want to maintain multiple environments, i.e., Testing separate from production.
- Offers proper isolation between environments allowing developers to maintain several versions of software in different environments.
- Suited for situations where you don't control the timing of the release

Pros and Cons of Truck-based Development:

- Developers work in a single branch called 'trunk' and make smaller changes more frequently.
- Quicker release - as multiple frequent changes allows features to be released much faster
- Less Merge Conflicts – as long-lived branches are non existent
- Better suited to senior developers - offers a great amount of autonomy that non-experienced developers may find difficult when interacting directly with the trunk.
- Can be messy as developers all interact directly with the trunk.