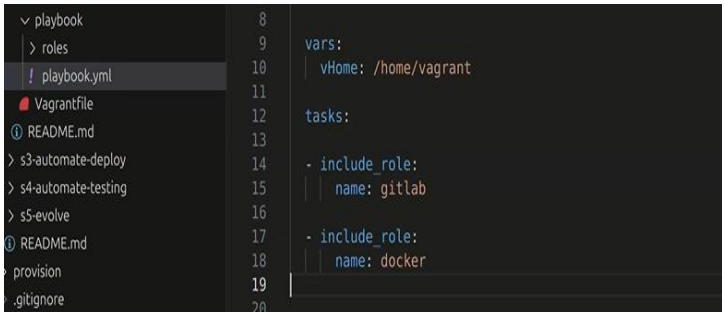
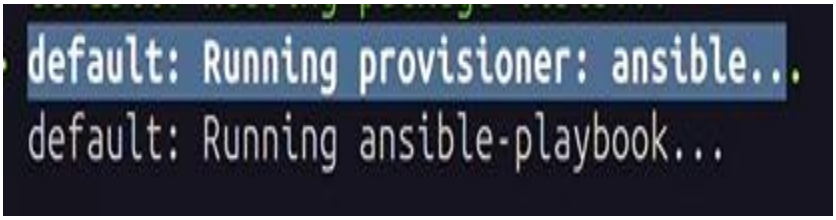
EXERCISE 1

1.a Analyse how the provisioning is done.

Answer:

- When vagrant up is run, it executes an Ansible **playbook** (playbook.yml).
- Provisioning is performed automatically by **Ansible**, an automation tool, which is triggered by Vagrant.
- This playbook contains a series of **tasks** that run on the virtual machine, such as installing dependencies, adding software repositories, and installing the GitLab and Docker packages without any manual intervention.



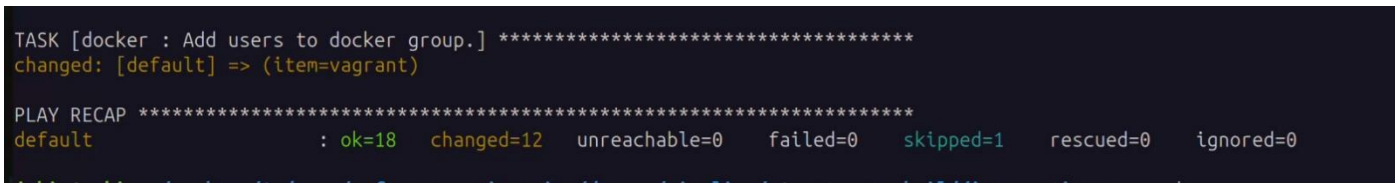


1.b Is there any "qualities" being addressed by the way the provisioning is done? Justify.

Answer:

Yes, several key software qualities are addressed:

- **Automation:** The entire server setup is scripted from start to finish, which eliminates the need for manual commands and reduces the chance of human error.
- **Repeatability & Consistency:** The Ansible playbook guarantees that the server is set up identically every single time it is created. This ensures every developer has a consistent environment.
- **Maintainability:** The server's configuration is defined as code in the playbook file. This makes it easy to read, track changes in version control (like Git), and modify the server setup in a documented way.

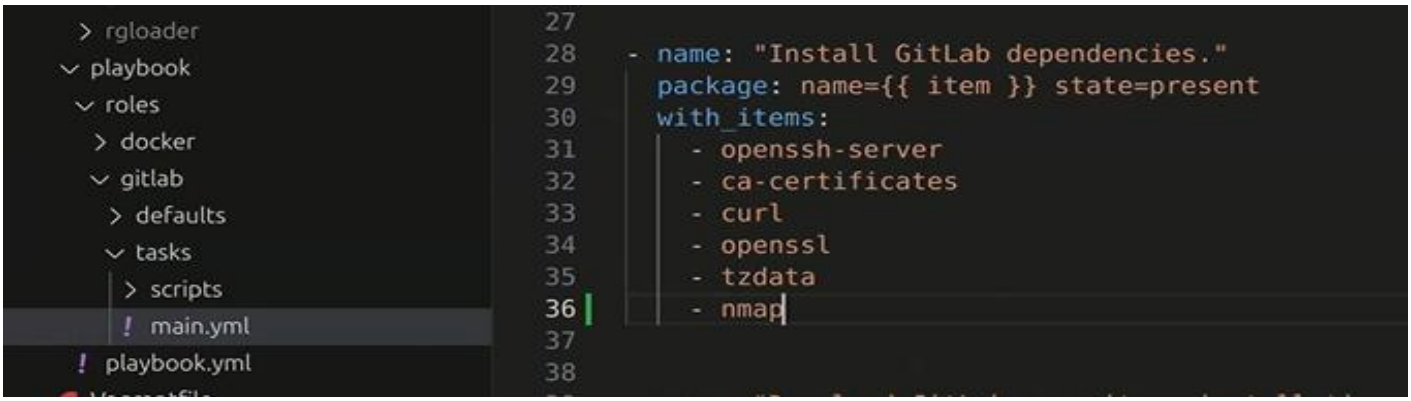


1.c Add package "nmap" to the playbook.

Answer:

This was achieved by:

1. Editing the Ansible playbook file (e.g., roles/gitlab/tasks/main.yml).
2. Adding nmap to the list of packages to be installed by the apt task (name: "Install GitLab dependencies." then Add - nmap to the list).
3. Re-running the provisioning with the command vagrant provision.
4. Verifying the installation by connecting to the VM with vagrant ssh and running nmap --version to confirm it was installed correctly.



EXERCISE 2

Exercise 2.a: Collect the manual steps you have done up to now.

Answer:

This lists the server configuration steps I did *after* the initial vagrant up command.

1. GitLab Configuration:

- Connected to the VM using vagrant ssh.
- Edited the /etc/gitlab/gitlab.rb file to set the external_url (first to 192.168.33.9, then troubleshooting and correcting it to 192.168.56.9).
- Also in /etc/gitlab/gitlab.rb, uncommented and changed puma['port'] to 8088.

- Ran sudo gitlab-ctl reconfigure and sudo gitlab-ctl restart to apply the changes.

2. Docker Permissions:

- Added the vagrant user to the docker group using sudo usermod -aG docker vagrant.
- Exited and re-connected with vagrant ssh to make the permission change take effect.

3. GitLab Runner Installation and Registration:

- Installed the GitLab Runner software using curl and sudo apt-get install gitlab-runner.
- Manually registered the runner by running sudo gitlab-runner register and interactively providing:
  - The GitLab instance URL.
  - The project-specific registration token.
  - A description, tags, the docker executor, and a default image.
- Restarted the runner with sudo gitlab-runner restart.
- Manually edited the runner in the GitLab web UI to enable **"Run untagged jobs"**.

Exercise 2.b: List those that could be added to a playbook.

Answer:

Almost every manual server configuration step can be automated with Ansible.

- **GitLab Configuration:** The changes to /etc/gitlab/gitlab.rb are perfect for automation.
- **Running Reconfigure:** Ansible can be configured to automatically run gitlab-ctl reconfigure only when the gitlab.rb file has actually changed. This is done using "handlers".
- **Docker Permissions:** Adding a user to a group is a standard task that Ansible's user module can handle easily.
- **GitLab Runner Installation:** Installing packages is one of Ansible's most basic and powerful features using the apt module.
- **GitLab Runner Registration:** This can also be automated. The gitlab-runner register command can be run in "non-interactive" mode with all the answers provided as command-line arguments. Ansible's command module can execute this.

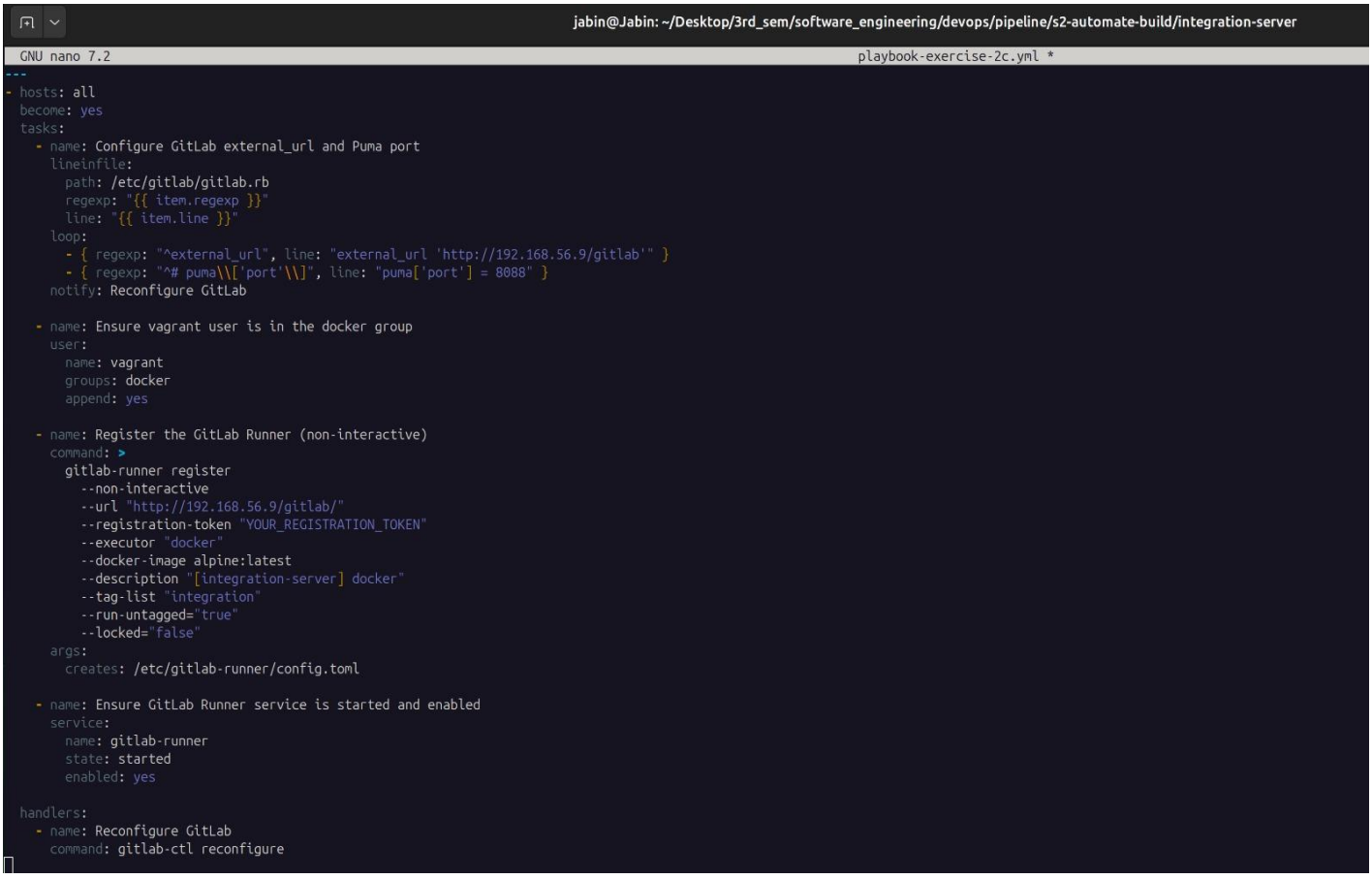
The only step that is difficult to automate with a simple Ansible playbook is clicking the checkbox in the web UI, as that would typically require using the GitLab API.

Exercise 2.c: Add (at least) half of them to a playbook.

Answer:

First, it configures the gitlab.rb file with the correct URL and port. Second, it ensures the vagrant user has the right permissions for Docker. Third, and most importantly, it automates the runner registration by running the command in non-interactive mode. This playbook even handles running gitlab-ctl reconfigure automatically if the configuration file changes. This approach is far more robust and repeatable than performing the steps manually.

Here is a code I added to my playbook.yml file to automate the GitLab configuration and Docker permissions.

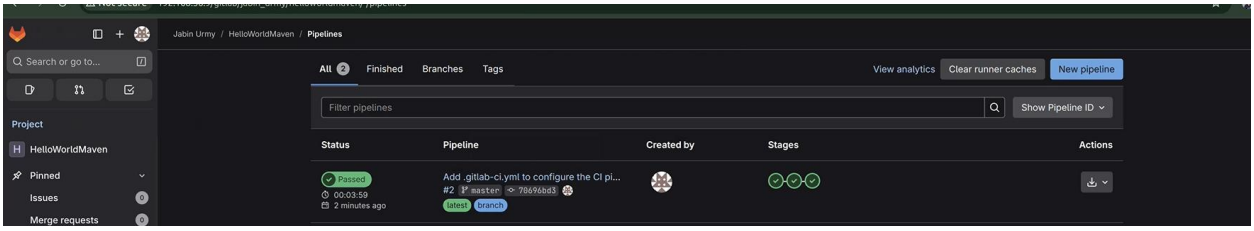


EXERCISE 3

3.a) Use GitLab to analyse what has just happened?

Answer:

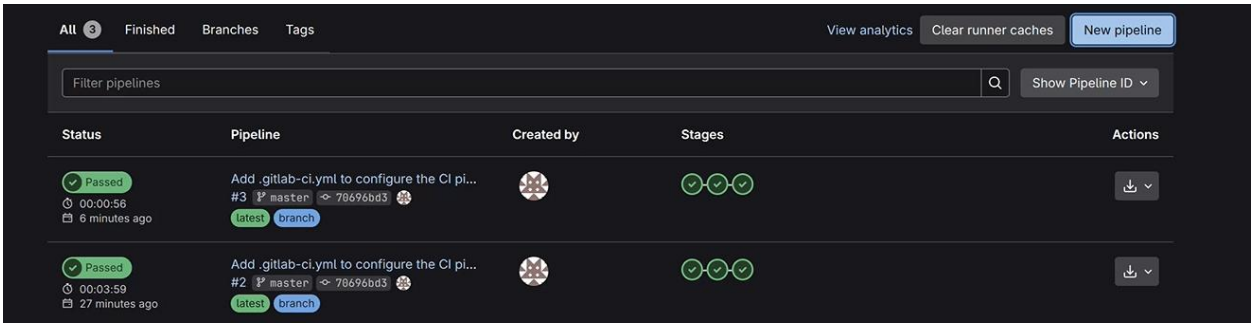
- When I pushed the .gitlab-ci.yml file to the repository, GitLab automatically detected it and triggered a new CI/CD pipeline.
- The pipeline followed the instructions in the file: it created a temporary Docker container using the maven:latest image.
- It then executed the three stages in order: build (running mvn compile), test (running mvn test), and run (running mvn package and mvn exec:java).



3.b) Try to run again the CI and inspect the intermediate results of each stage.

Answer:

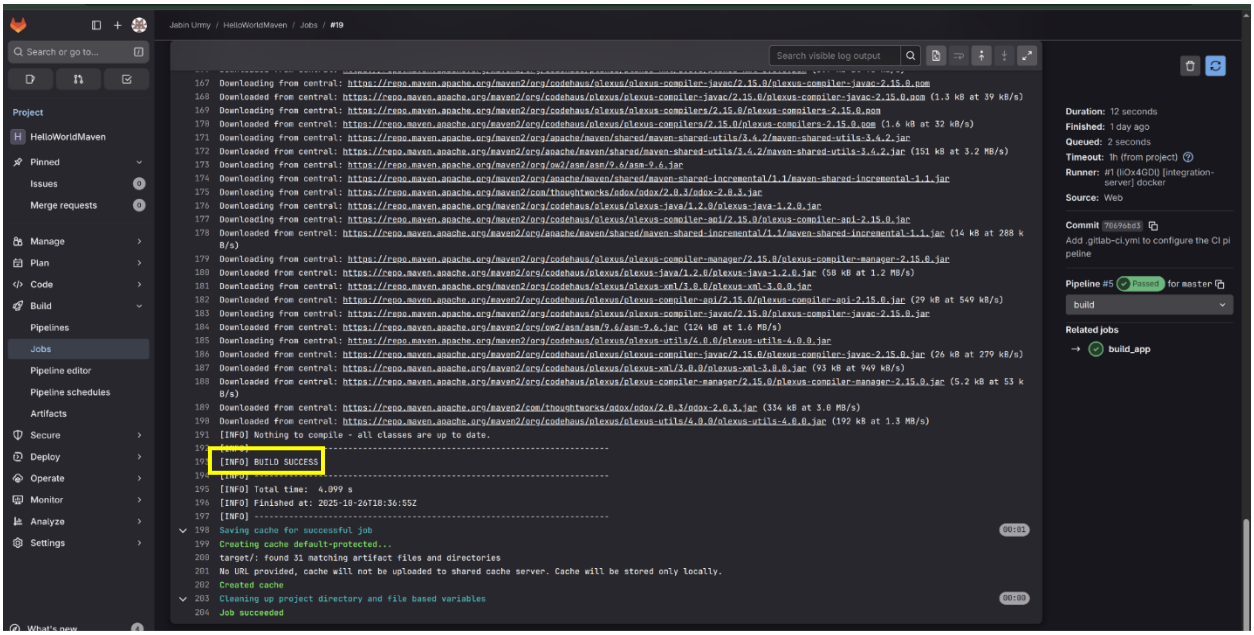
- To run the pipeline again, I went to the **Build -> Pipelines** page and click the blue **"Run pipeline"** button.
- To inspect the results, I checked on the pipeline's status icon (e.g., the green checkmark). This shows the individual jobs (build_app, test_app, run_app).



Build app:

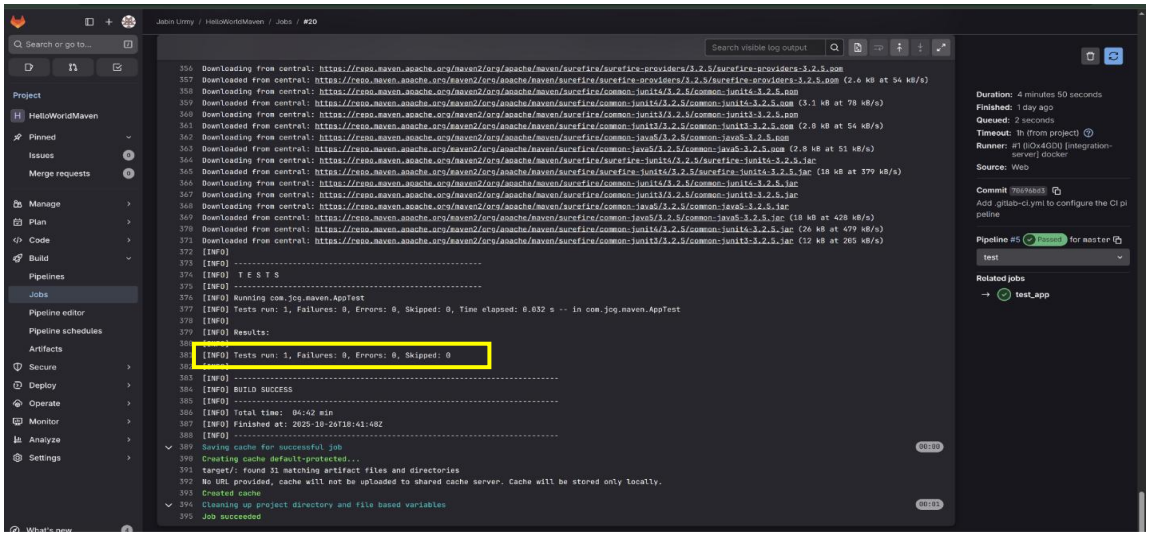
- The log shows Maven successfully downloading all the necessary dependencies from the central repository.

- The stage concludes with the message **[INFO] BUILD SUCCESS**, which is the intermediate result proving that the Java source code compiled without any errors.



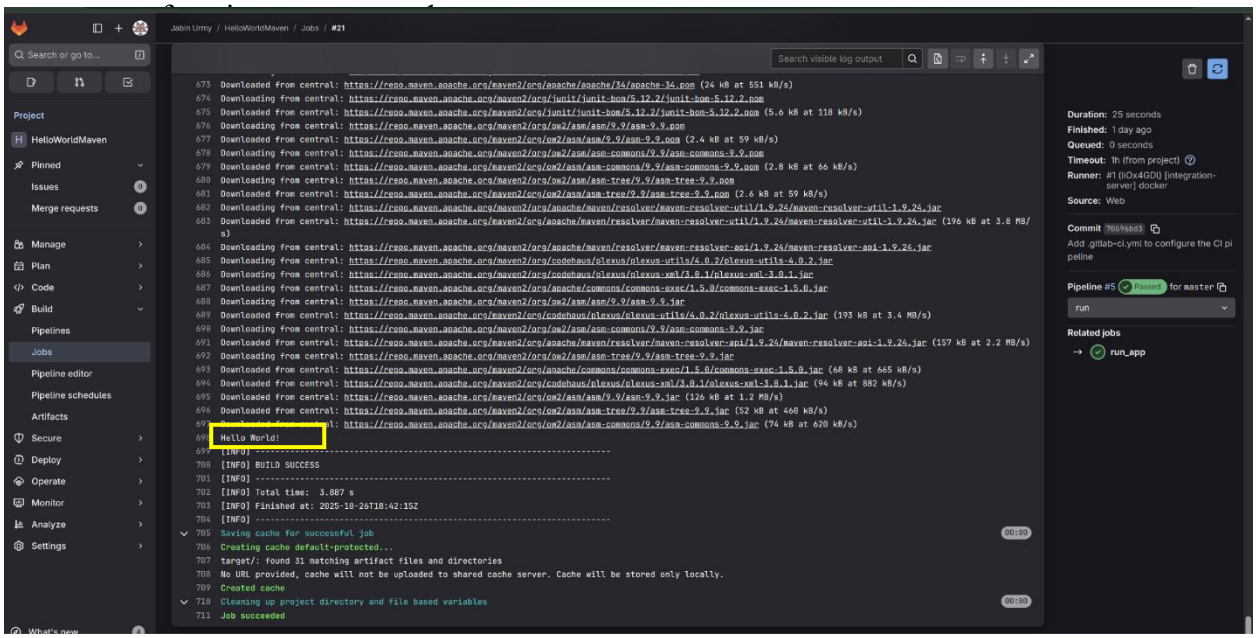
Test app:

- The log shows the Maven Surefire Plugin executing the unit tests.
- The summary at the end, **[INFO] Results: Tests run: 1, Failures: 0, Errors: 0, Skipped: 0**, is the intermediate result. It confirms that all automated tests passed, ensuring the code quality meets the requirements.



Run app:

- This log shows two key results. First, Maven packages the compiled code into a distributable file, indicated by the message **[INFO] Building jar: ...**.
- Second, the application is executed directly on the runner, and its output, **Hello World!**, is printed to the console. This proves that the packaged application is executable and



3.c) Where is the .jar file generated as result of the build?

Answer:

The .jar file is generated in the target/ directory during the run_app job.

Because the .gitlab-ci.yml file includes a cache: directive for the target/ path, the GitLab Runner saves this directory's contents after the job completes.

This cache is stored on the server inside a Docker volume managed by the GitLab Runner, located at a path similar to /var/lib/docker/volumes/.../_data/.

Crucially, while the file is saved, it is not a "build artifact." Its purpose is to speed up subsequent pipeline runs, not for users to download. The file is not easily accessible from the GitLab UI at this stage.

EXERCISE 4

Exercise 4.a: Find out what is the result of this modification.

Answer:

A deploy Stage and artifacts Keyword were Added: The .gitlab-ci.yml was modified to include a new deploy stage. This stage uses the artifacts keyword, which is specifically designed to capture and store the final products of a job.

It Solves the Problem from Exercise 3: This is fundamentally different from the cache. The cache is for temporary storage to speed up future jobs, while artifacts are for permanently preserving the valuable output of a pipeline for users.

The Result is a Downloadable, Working Application:

The most visible result is the download button now available on the completed pipeline page in GitLab.

I verified the entire end-to-end process by downloading the artifact, extracting the .jar file, and executing it.

The successful execution (cowsay Hello World) provides definitive proof that the CI pipeline is correctly building, testing, and packaging a valid, distributable application.