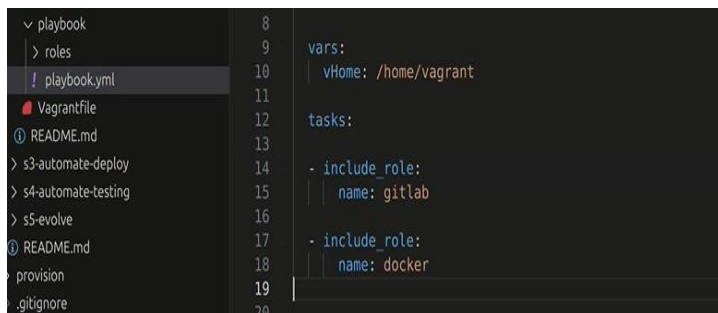


## EXERCISE 1

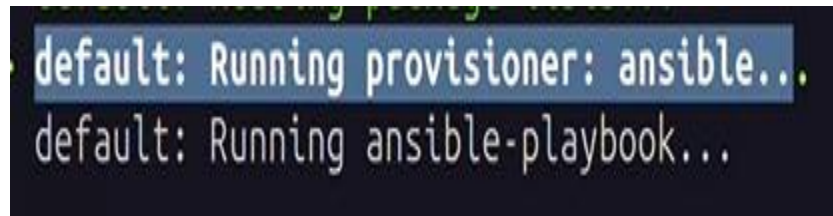
### 1.a Analyse how the provisioning is done.

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.



```
8
9 vars:
10   vHome: /home/vagrant
11
12 tasks:
13
14   - include role:
15     name: gitlab
16
17   - include role:
18     name: docker
19
20
```

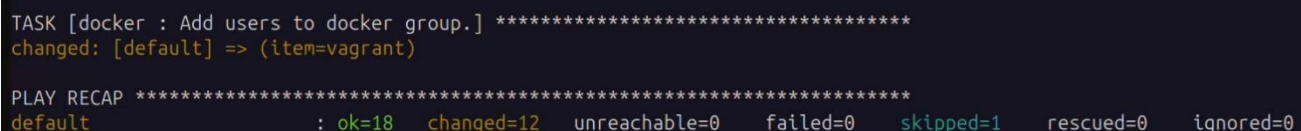


### 1.b Is there any "qualities" being addressed by the way the provisioning is done? Justify.

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.



```
TASK [docker : Add users to docker group.] *****
changed: [default] => (item=vagrant)

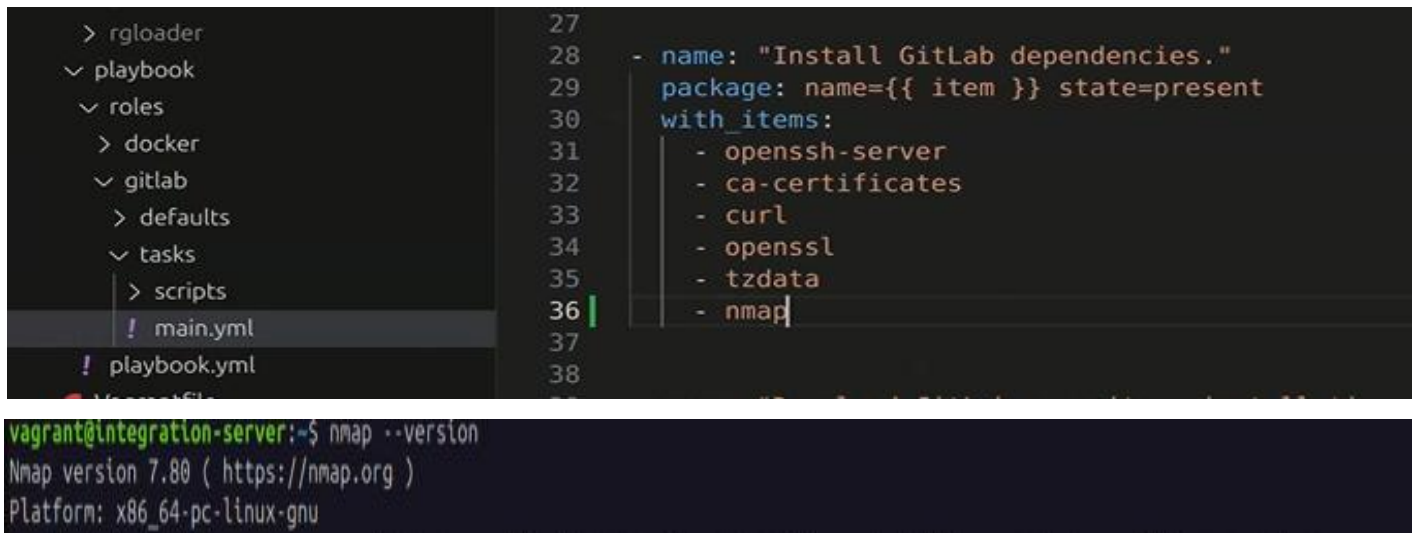
PLAY RECAP *****
default : ok=18  changed=12  unreachable=0  failed=0  skipped=1  rescued=0  ignored=0
```

### 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.



The image shows a terminal window with two parts. The top part displays the Ansible playbook file structure in a tree view on the left and the content of the `main.yml` file on the right. The bottom part shows a terminal prompt where the `nmap --version` command was executed, displaying the version and platform information.

```
> rgloader
├── playbook
│   └── roles
│       ├── docker
│       └── gitlab
│           ├── defaults
│           └── tasks
│               ├── scripts
│               └── ! main.yml
└── ! playbook.yml
```

```
27
28 - name: "Install GitLab dependencies."
29   package: name={{ item }} state=present
30   with_items:
31     - openssh-server
32     - ca-certificates
33     - curl
34     - openssl
35     - tzdata
36     - nmap
37
38
```

```
vagrant@integration-server:~$ nmap --version
Nmap version 7.80 ( https://nmap.org )
Platform: x86_64-pc-linux-gnu
```

## 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.

```
jabin@Jabin: ~/Desktop/3rd_sem/software_engineering/devops/pipeline/s2-automate-build/integration-server
GNU nano 7.2                                playbook-exercise-2c.yml *
- hosts: all
  become: yes
  tasks:
    - name: Configure GitLab external_url and Puma port
      lineinfile:
        path: /etc/gitlab/gitlab.rb
        regexp: "{{ item.regexp }}"
        line: "{{ item.line }}"
      loop:
        - { regexp: "^external_url", line: "external_url 'http://192.168.56.9/gitlab'" }
        - { regexp: "^# puma\\[\\['port'\\]\\]", line: "puma['port'] = 8088" }
      notify: Reconfigure GitLab

    - name: Ensure vagrant user is in the docker group
      user:
        name: vagrant
        groups: docker
        append: yes

    - name: Register the GitLab Runner (non-interactive)
      command: >
        gitlab-runner register
        --non-interactive
        --url "http://192.168.56.9/gitlab/"
        --registration-token "YOUR_REGISTRATION_TOKEN"
        --executor "docker"
        --docker-image alpine:latest
        --description "[integration-server] docker"
        --tag-list "integration"
        --run-untagged="true"
        --locked="false"
      args:
        creates: /etc/gitlab-runner/config.toml

    - name: Ensure GitLab Runner service is started and enabled
      service:
        name: gitlab-runner
        state: started
        enabled: yes

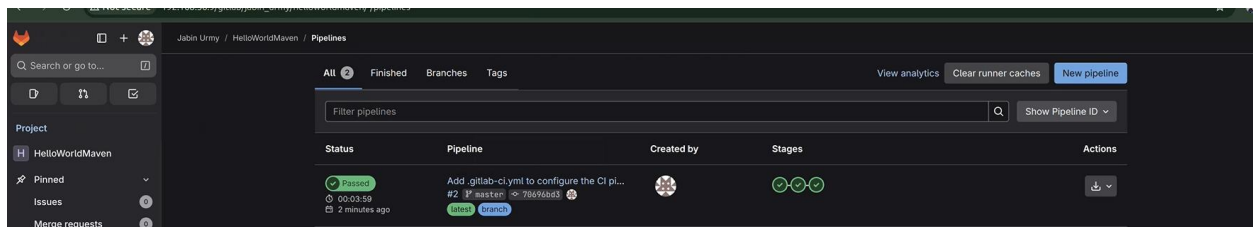
  handlers:
    - name: Reconfigure GitLab
      command: gitlab-ctl reconfigure
```

## EXERCISE 3

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

Answer:

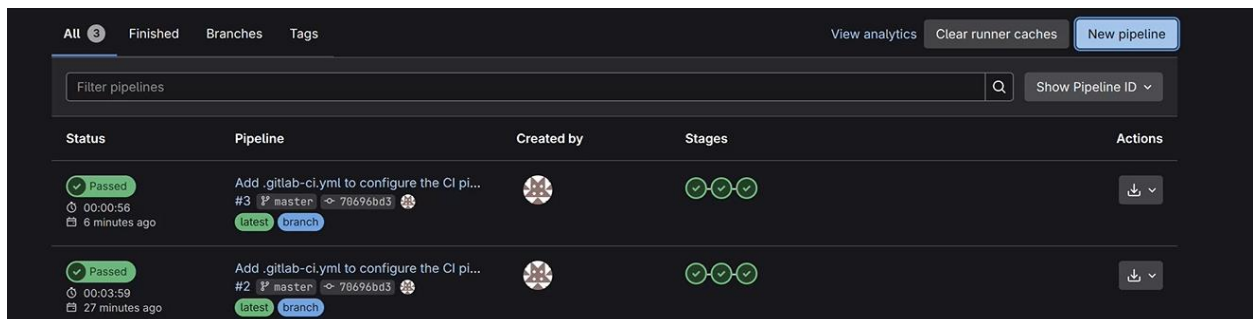
- When I pushed the `.gitlab-ci.yml` file to the repository, GitLab automatically detected it and triggered a new CI/CD pipeline.
- The pipeline followed the instructions in the file: it created a temporary Docker container using the `maven:latest` image.
- It then executed the three stages in order: 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:

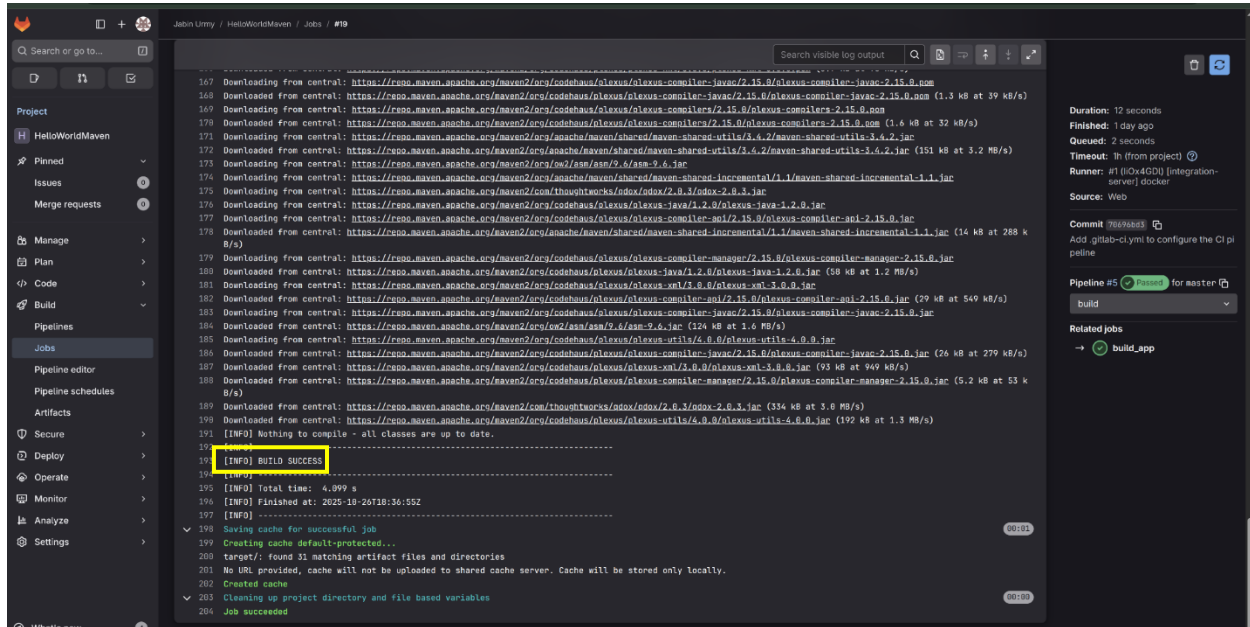
- To run the pipeline again, I went to the **Build -> Pipelines** page and click the blue **"Run pipeline"** button.
- To inspect the results, I checked on the pipeline's status icon (e.g., the green checkmark). This shows the individual jobs (`build_app`, `test_app`, `run_app`).



### Build app:

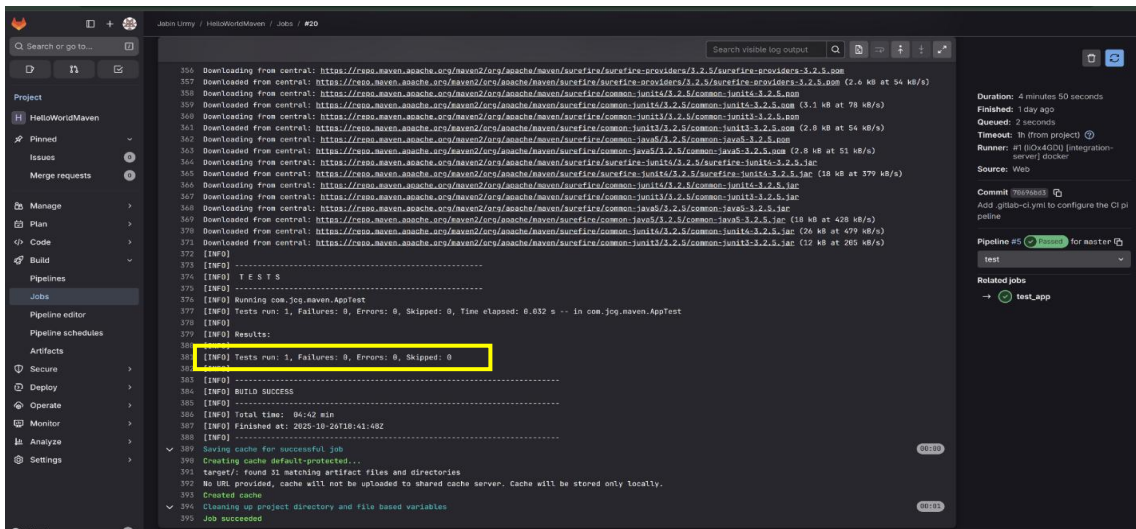
- The log shows Maven successfully downloading all the necessary dependencies from the central repository.

- 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:

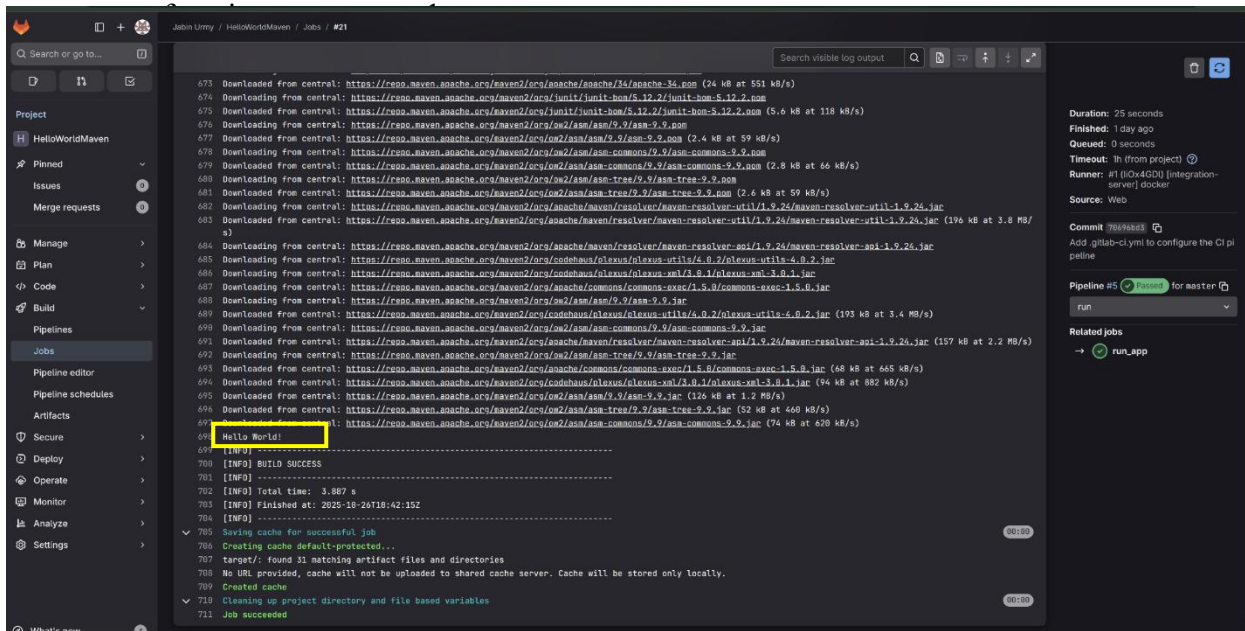
- The log shows the Maven Surefire Plugin executing the unit tests.
- The summary at the end, **[INFO] Results: Tests run: 1, Failures: 0, Errors: 0, Skipped: 0**, is the intermediate result. It confirms that all automated tests passed, ensuring the code quality meets the requirements.





## 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



```
673 Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/apache/34/apache-34.jar (24 kB at 551 kB/s)
674 Downloaded from central: https://repo.maven.apache.org/maven2/org/junit/junit-bom/5.12.2/junit-bom-5.12.2.pom
675 Downloaded from central: https://repo.maven.apache.org/maven2/org/junit/junit-bom/5.12.2/junit-bom-5.12.2.pom (5.6 kB at 118 kB/s)
676 Downloaded from central: https://repo.maven.apache.org/maven2/org/junit/junit-bom/5.12.2/junit-bom-5.12.2.pom (5.6 kB at 118 kB/s)
677 Downloaded from central: https://repo.maven.apache.org/maven2/org/junit/junit-bom/5.12.2/junit-bom-5.12.2.pom (5.6 kB at 118 kB/s)
678 Downloaded from central: https://repo.maven.apache.org/maven2/org/junit/junit-bom/5.12.2/junit-bom-5.12.2.pom (5.6 kB at 118 kB/s)
679 Downloaded from central: https://repo.maven.apache.org/maven2/org/junit/junit-bom/5.12.2/junit-bom-5.12.2.pom (5.6 kB at 118 kB/s)
680 Downloaded from central: https://repo.maven.apache.org/maven2/org/junit/junit-bom/5.12.2/junit-bom-5.12.2.pom (5.6 kB at 118 kB/s)
681 Downloaded from central: https://repo.maven.apache.org/maven2/org/junit/junit-bom/5.12.2/junit-bom-5.12.2.pom (5.6 kB at 118 kB/s)
682 Downloaded from central: https://repo.maven.apache.org/maven2/org/junit/junit-bom/5.12.2/junit-bom-5.12.2.pom (5.6 kB at 118 kB/s)
683 Downloaded from central: https://repo.maven.apache.org/maven2/org/junit/junit-bom/5.12.2/junit-bom-5.12.2.pom (5.6 kB at 118 kB/s)
684 Downloaded from central: https://repo.maven.apache.org/maven2/org/junit/junit-bom/5.12.2/junit-bom-5.12.2.pom (5.6 kB at 118 kB/s)
685 Downloaded from central: https://repo.maven.apache.org/maven2/org/junit/junit-bom/5.12.2/junit-bom-5.12.2.pom (5.6 kB at 118 kB/s)
686 Downloaded from central: https://repo.maven.apache.org/maven2/org/junit/junit-bom/5.12.2/junit-bom-5.12.2.pom (5.6 kB at 118 kB/s)
687 Downloaded from central: https://repo.maven.apache.org/maven2/org/junit/junit-bom/5.12.2/junit-bom-5.12.2.pom (5.6 kB at 118 kB/s)
688 Downloaded from central: https://repo.maven.apache.org/maven2/org/junit/junit-bom/5.12.2/junit-bom-5.12.2.pom (5.6 kB at 118 kB/s)
689 Downloaded from central: https://repo.maven.apache.org/maven2/org/junit/junit-bom/5.12.2/junit-bom-5.12.2.pom (5.6 kB at 118 kB/s)
690 Downloaded from central: https://repo.maven.apache.org/maven2/org/junit/junit-bom/5.12.2/junit-bom-5.12.2.pom (5.6 kB at 118 kB/s)
691 Downloaded from central: https://repo.maven.apache.org/maven2/org/junit/junit-bom/5.12.2/junit-bom-5.12.2.pom (5.6 kB at 118 kB/s)
692 Downloaded from central: https://repo.maven.apache.org/maven2/org/junit/junit-bom/5.12.2/junit-bom-5.12.2.pom (5.6 kB at 118 kB/s)
693 Downloaded from central: https://repo.maven.apache.org/maven2/org/junit/junit-bom/5.12.2/junit-bom-5.12.2.pom (5.6 kB at 118 kB/s)
694 Downloaded from central: https://repo.maven.apache.org/maven2/org/junit/junit-bom/5.12.2/junit-bom-5.12.2.pom (5.6 kB at 118 kB/s)
695 Downloaded from central: https://repo.maven.apache.org/maven2/org/junit/junit-bom/5.12.2/junit-bom-5.12.2.pom (5.6 kB at 118 kB/s)
696 Downloaded from central: https://repo.maven.apache.org/maven2/org/junit/junit-bom/5.12.2/junit-bom-5.12.2.pom (5.6 kB at 118 kB/s)
697 Downloaded from central: https://repo.maven.apache.org/maven2/org/junit/junit-bom/5.12.2/junit-bom-5.12.2.pom (5.6 kB at 118 kB/s)
698 Downloaded from central: https://repo.maven.apache.org/maven2/org/junit/junit-bom/5.12.2/junit-bom-5.12.2.pom (5.6 kB at 118 kB/s)
699 Hello World!
700 [INFO] BUILD SUCCESS
701 [INFO]
702 [INFO] Total time: 3.887 s
703 [INFO] Finished at: 2025-10-26T18:42:15Z
704 [INFO]
705 Saving cache for successful job
706 Creating cache default-protected...
707 target/: found 31 matching artifact files and directories
708 No URL provided, cache will not be uploaded to shared cache server. Cache will be stored only locally.
709 Created cache
710 Cleaning up project directory and file based variables
711 Job succeeded
```

## 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.

The screenshot shows the GitLab CI/CD interface for a pipeline named `deploy_app`. The pipeline is in a `Completed` state, having finished 1 day ago. The left sidebar shows the project structure, including `Jobs` and `Pipelines`. The main panel displays the pipeline's execution log, which includes steps for preparing the Docker executor, pulling the Docker image, running the container, and finally deploying the application. The right sidebar provides summary information: Duration: 6 seconds, Finished: 1 day ago, Queued: 3 seconds, Timeout: 1h (from project), Runner: #1 (Ox4GD) [Integration-server] Docker, Source: Push, Job artifacts: These artifacts are the latest. They will not be deleted (even if expired) until newer artifacts are available. The pipeline is associated with commit `738c4c3d` and the `deploy` stage. The `deploy_app` job is shown as successful.

```
1 Running with gitlab-runner 18.5.0 (bda6871)
2 on [integration-server] docker 110x40D13, system ID: s_1h12ad2317a0
3 Preparing the "docker" executor
4 Using Docker executor with image maven:latest ...
5 Using effective pull policy of [always] for container maven:latest
6 Pulling docker image maven:latest ...
7 Using docker image sha256:daad458da4702d6db8450c7e7c076944f04e78078acfb82747bd2deeb86ca83 for maven:latest with digest maven@sha256:3d358950456f61758a4b3f62c7460ff6369d5cabefec04ee4387e76dc6440972 ...
8 Preparing environment
9 Using effective pull policy of [always] for container sha256:9a2874a12c7d1ef6a13167cdee5a465ef8c2b548c510e78bdf1c71d07213279
10 Running on runner-110x40D13-project-1-concurrent-0 via integration-server...
11 Setting source from Git repository
12 Gitly correlation ID: 01K8G26MNGF66A4YVCHWBJM
13 Fetching changes with git depth set to 20...
14 Reinitialized existing Git repository in /builds/gitlab/jobin_urny/helloworldmaven/.git/
15 Created fresh repository.
16 Checking out 738c4c3d as detached HEAD (ref is master)...
17 Removing target/
18 Skipping Git submodules setup
19 Restoring cache
20 Checking cache for default-protected...
21 No URL provided, cache will not be downloaded from shared cache server. Instead a local version of cache will be extracted.
22 Successfully extracted cache
23 Executing "stop_script" stage of the job script
24 Using effective pull policy of [always] for container maven:latest
25 Using docker image sha256:daad458da4702d6db8450c7e7c076944f04e78078acfb82747bd2deeb86ca83 for maven:latest with digest maven@sha256:3d358950456f61758a4b3f62c7460ff6369d5cabefec04ee4387e76dc6440972 ...
26 $ echo "Deploy review app"
27 Deploy review app
28 Saving cache for successful job
29 Creating cache default-protected...
30 target/: found 31 matching artifact files and directories
31 Archive is up to date!
32 Created cache
33 Uploading artifacts for successful job
34 Uploading artifacts...
35 target/*: found 1 matching artifact files and directories
36 Uploading artifacts as "archive" to coordinator... 201 Created correlation_id=01K8G2727Y6P426F8C3081MIF id=25 responseStatus=201 Created token=64_Ha5pZ
37 Cleaning up project directory and file based variables
38 Job succeeded
```