Discovery 9: Integrate Application into Existing CI/CD Environment

Task 1: Examine Provided Application

In this procedure, you will examine the application that was already prepared. The code for the application is stored in GitLab. You will clone and test the application.

Activity

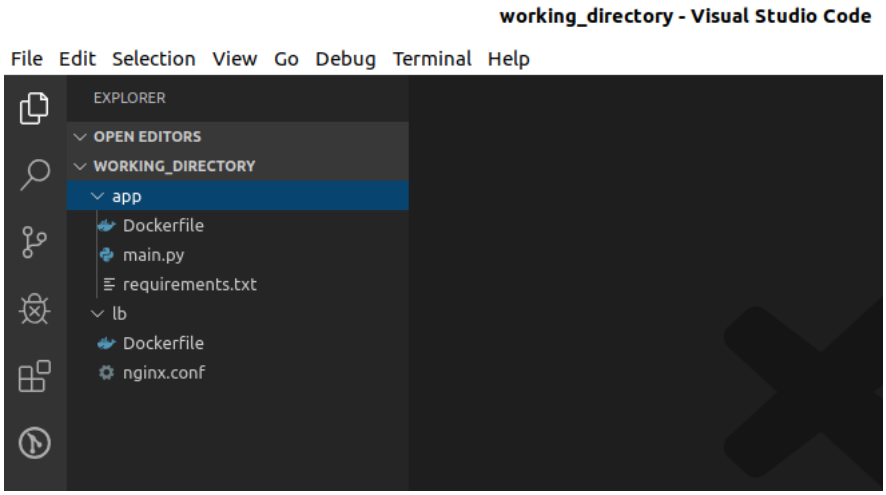
Step 1: Open the web browser on the Student VM and connect to GitLab. Use the *http://dev.gitlab.local* URL.

Step 2: Log in to GitLab. Use credentials *root* / *1234QWer*.

Step 3: Copy the HTTP link by clicking the icon next to the URL.

Step 4: Open Visual Studio Code and open the *working_directory* folder. Open the terminal and clone the Git repository from the link that you have copied in GitLab web user interface.

```
student@student-workstation:~/working_directory$ git clone http://dev.gitlab.local/root/application.git .
Cloning into '.'...
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 9 (delta 0), reused 9 (delta 0)
Unpacking objects: 100% (9/9), done.
```



You should see the files that are used to deploy the sample application. There are two main parts of the application: the application itself and the load balancer.

Step 5: Examine the application files.

app > main.py > ...

student, 6 hours ago | 1 author (student)

```
from flask import Flask
import socket

ip = socket.gethostbyname(socket.gethostname())

app = Flask(__name__)

@app.route('/')
def home():
    out = (
        f'Welcome to Cisco DevNet.<br>'
        f'IP address of the server is {ip}.<br><br>'
    )
    return out

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0')
```

app > Dockerfile

student, 6 hours ago | 1 author (student)

```
FROM python:3.7

COPY . /app
WORKDIR /app

RUN pip install -r requirements.txt
EXPOSE 5000

CMD ["python3", "main.py"]
```

app > requirements.txt

student, 6 hours ago | 1 author (student)

```
Flask==1.1.1
```

The files define a simple Flask application that displays the text in the browser. It also retrieves the IP address of the host where the application is running and displays the IP address in the web browser.

The application also comes with the **Dockerfile** file, which defines the Docker container for the application. The **requirements.txt** file is used to install all required Python packages.

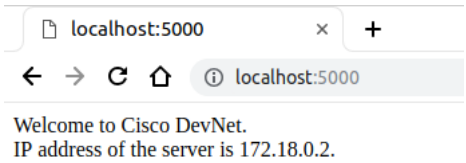
Step 6: Build the *app* container using the **docker build -t app .** command in the *app* folder.

```
student@student-workstation:~/working_directory$ cd app/
student@student-workstation:~/working_directory/app$ docker build -t app .
Sending build context to Docker daemon  4.096kB
<... output omitted ...>
Successfully built 179e804ccb12
Successfully tagged app:latest
```

Step 7: Run the Docker container to test the application. Use the **docker run -p 5000:5000 app** command.

```
student@student-workstation:~/working_directory/app$ docker run -p 5000:5000 app
 * Serving Flask app "main" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 247-145-030
```

Step 8: Open the web browser and access the application by using the *http://localhost:5000* URL.



You should see a similar output to this one. The output confirms that application is working. Press **Ctrl-C** in the terminal to stop the Docker container.

Step 9: Examine the load balancer files.

lb > Dockerfile

student, 6 hours ago | 1 author (student)

```
1  FROM nginx
2
3  COPY nginx.conf /etc/nginx/nginx.conf
4
5  EXPOSE 8080
6
7  CMD ["nginx", "-g", "daemon off;"]
8
9
```

lb > nginx.conf

student, 6 hours ago | 1 author (student)

```
1  events {}
2  http {
3
4    upstream myapp {
5      server 172.20.0.100:5000;
6      server 172.20.0.101:5000;
7    }
8
9    server {
10     listen 8080;
11     server_name localhost;
12
13     location / {
14       proxy_pass http://myapp;
15       proxy_set_header Host $host;
16     }
17   }
18
19 }
20
21
```

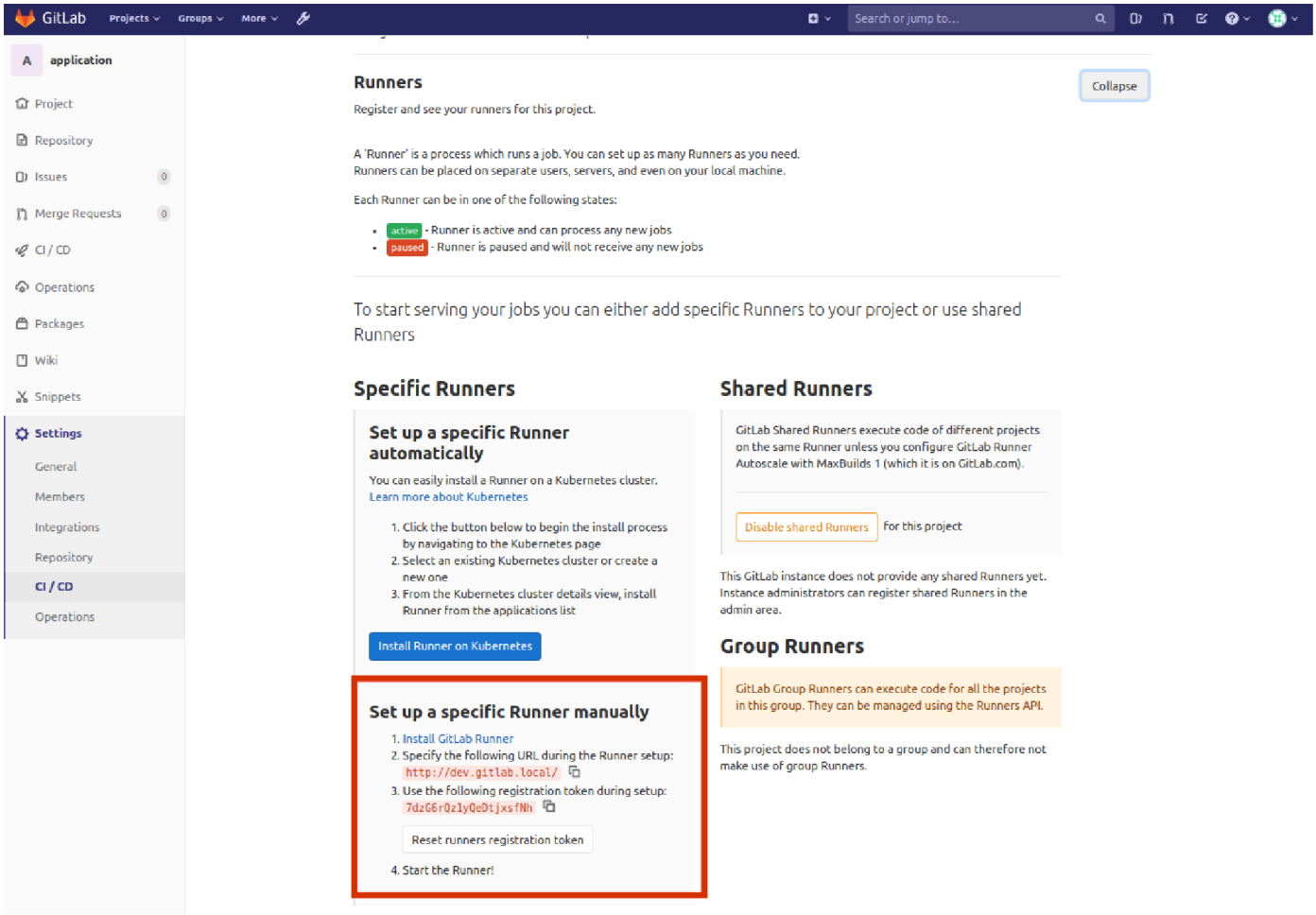
You should see the *nginx* configuration file. The configuration file defines two back-end servers. The load balancer will listen on the TCP port 8080. There is also the *Dockerfile* file, which defines the container for the load balancer.

Task 2: Prepare CI/CD Environment

To use the CI/CD environment in GitLab, you need to define a GitLab runner. You will use your student workstation as a runner. You will also add a simple CI/CD job to test that the runner is working as expected.

Activity

Step 1: In the GitLab web user interface, navigate to *Settings* > *CI/CD*. Expand the *Runners* section.



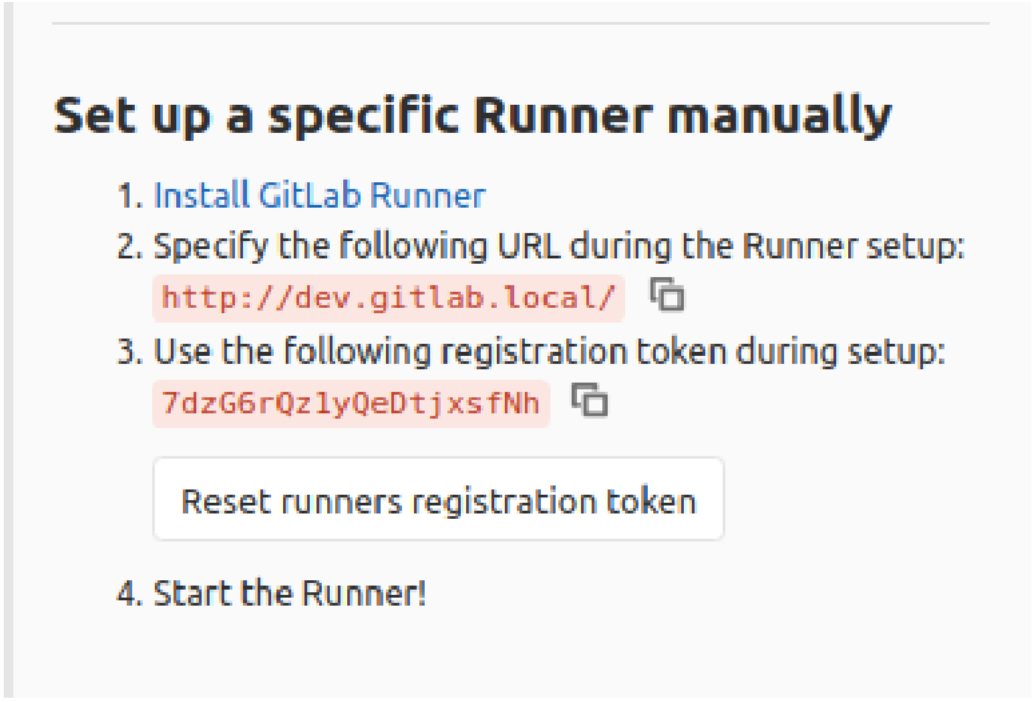
You should see the setup instructions for the GitLab runner.

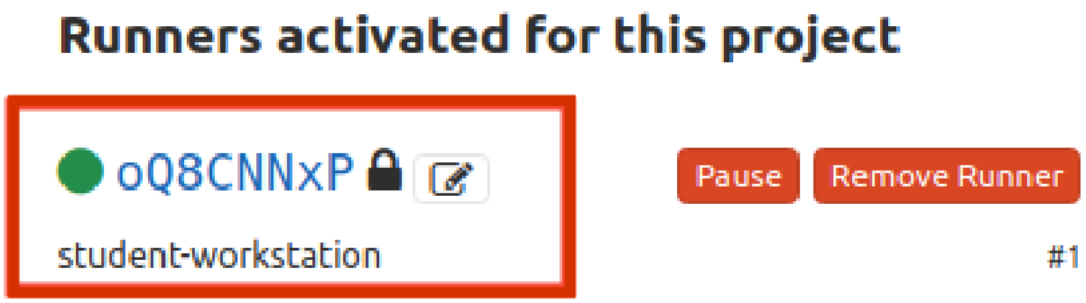
Step 2: Register the GitLab runner on the student workstation. In new terminal, use the **gitlab-runner register** command. Check the URL and the token in the GitLab web user interface. Choose *shell* for the executor. Leave everything else as the default values. Once done, start GitLab runner with the **gitlab-runner run -d /tmp** command.

```
student@student-workstation:~/working_directory/app$ gitlab-runner register
Runtime platform                                    arch=amd64 os=linux pid=31920 revision=05161b14 version=1
WARNING: Running in user-mode.
WARNING: The user-mode requires you to manually start builds processing:
WARNING: $ gitlab-runner run
WARNING: Use sudo for system-mode:
WARNING: $ sudo gitlab-runner...

Please enter the gitlab-ci coordinator URL (e.g. https://gitlab.com/):
http://dev.gitlab.local
Please enter the gitlab-ci token for this runner:
Vd2h3gxX7qFFazP_FWnh
Please enter the gitlab-ci description for this runner:
[student-workstation]:
Please enter the gitlab-ci tags for this runner (comma separated):

Registering runner... succeeded                     runner=Vd2h3gxX
Please enter the executor: custom, docker-ssh, kubernetes, docker, parallels, shell, ssh, virtualbox, docker+
shell
Runner registered successfully. Feel free to start it, but if it's running already the config should be autom
student@student-workstation:~/working_directory/app$ gitlab-runner run -d /tmp
Runtime platform                                    arch=amd64 os=linux pid=4205 revision=577f813d version=12
Starting multi-runner from /home/student/.gitlab-runner/config.toml ...  builds=0
WARNING: Running in user-mode.
WARNING: Use sudo for system-mode:
WARNING: $ sudo gitlab-runner...

Configuration loaded                                builds=0
Locking configuration file                          builds=0 file=/home/student/.gitlab-runner/config.toml pi
listen_address not defined, metrics & debug endpoints disabled  builds=0
[session_server].listen_address not defined, session endpoints disabled  builds=0
Checking for jobs... received                       job=1 repo_url=http://dev.gitlab.local/root/application.g
WARNING: Job failed: exit status 1                  duration=555.856965ms job=1 project=1 runner=Vd2h3gxX
WARNING: Failed to process runner                   builds=0 error=exit status 1 executor=shell runner=Vd2h3g
```

You should see that the GitLab runner was registered successfully and that it is running.

Step 3: Refresh the CI/CD page on the GitLab web user interface. Verify that the GitLab runner is successfully registered.





You should see the green dot next to the runner definition. It means that the runner is successfully registered to the GitLab.

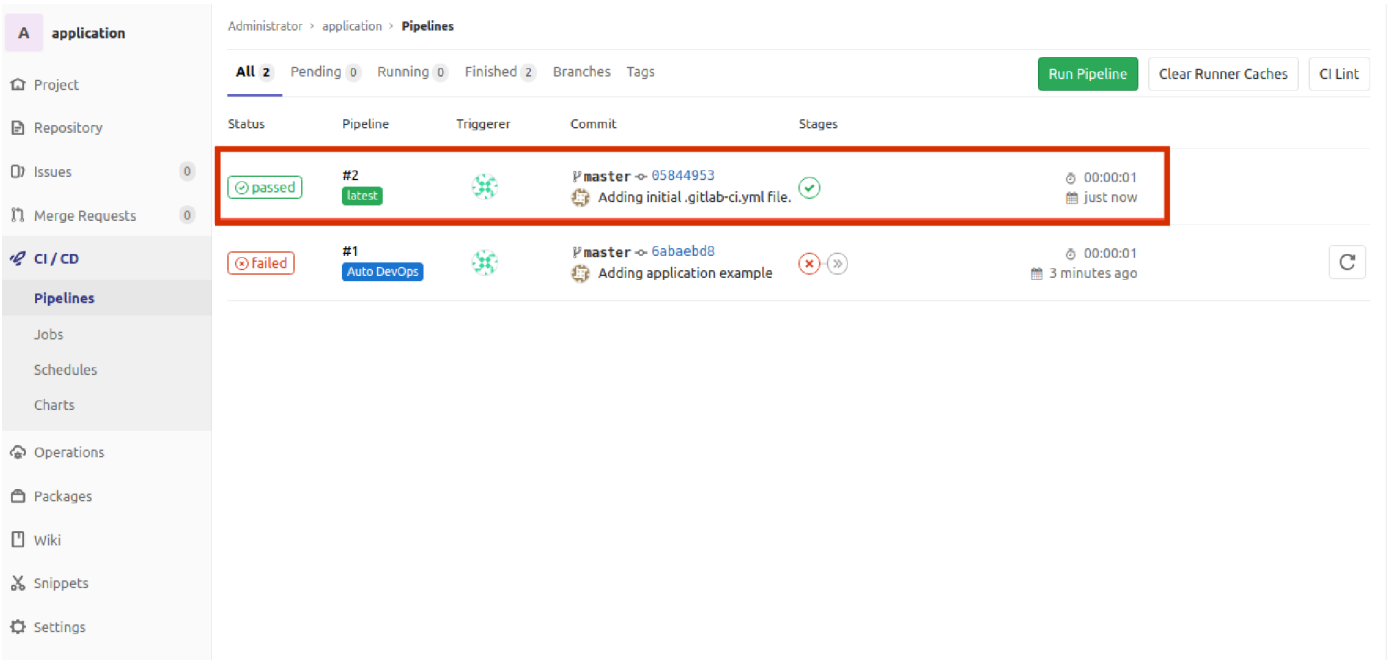
Step 4: Create a new file in the application folder. The name of the file should be *.gitlab-ci.yml*. Add the following content to the file to test that the runner is working:

```
job1:
  script: echo "Running test CI/CD."
```

Step 5: Commit the file and push the file to the repository. Use the *root/1234QWer* credentials for authentication.

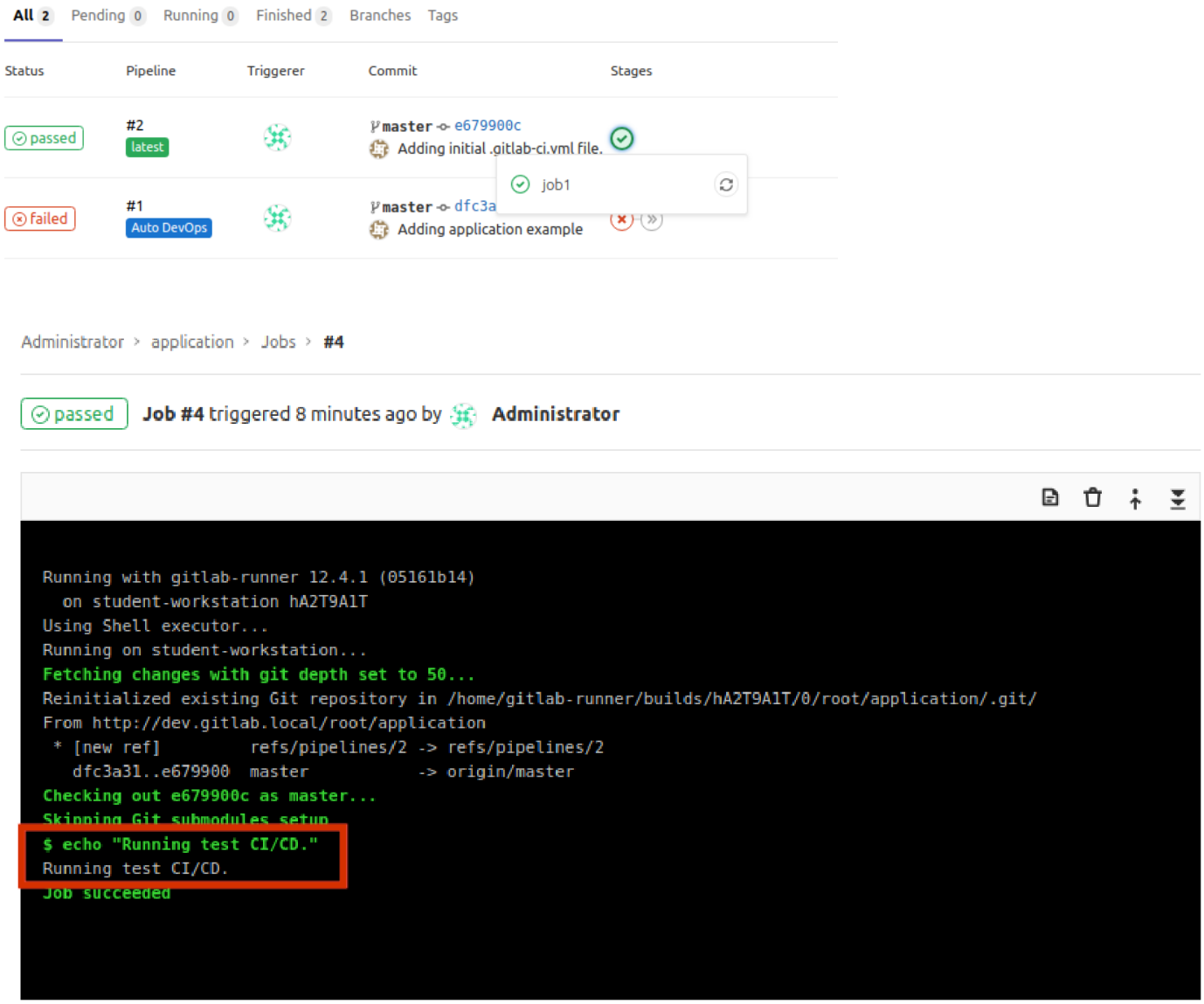
```
student@student-workstation:~/working_directory$ git status
On branch master
Your branch is up to date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .gitlab-ci.yml

nothing added to commit but untracked files present (use "git add" to track)
student@student-workstation:~/working_directory$ git add .gitlab-ci.yml
student@student-workstation:~/working_directory$ git commit -am "Adding initial .gitlab-ci.yml file."
[master e679900] Adding initial .gitlab-ci.yml file.
 1 file changed, 2 insertions(+)
 create mode 100644 .gitlab-ci.yml
student@student-workstation:~/working_directory$ git push
Username for 'http://dev.gitlab.local': root
Password for 'http://root@dev.gitlab.local': 1234QWer
Counting objects: 3, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 358 bytes | 358.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To http://dev.gitlab.local/root/application.git
   dfc3a31..e679900  master -> master
```

Step 6: In the GitLab web user interface, navigate to **CI/CD > Pipelines**. Observe the pipelines.



You should see that your first pipeline was executed. GitLab automatically executes the pipeline when a *.gitlab-ci.yml* file is located in the root of the repository.

Step 7: Click the green tick and select the job. Observe the job output.



You should see the output that you have defined in your *.gitlab-ci.yml* file under the *script* option. You have successfully set up the CI/CD environment.

Task 3: Implement Build Stage

Now you will define the build stage in the CI/CD pipeline. The build stage will be used to build an application container.

Activity

Step 1: Delete everything from the *.gitlab-ci.yml* file and add the following content to the file:

```
stages:
  - build

build:
  stage: build
  script: cd $CI_PROJECT_DIR/app && docker build -t app .
```

The specified configuration in the *.gitlab-ci.yml* file defines the CI/CD pipeline. You can configure CI/CD pipeline to be executed in multiple stages.

The stages section defines the stages in the CI/CD pipeline. In this particular example, there is only one stage.

Jobs define the actions that will be executed. Each job has a name. In the example, you are using the name *build*. You can map the job to the pipeline stage with the *stage* option. The script option defines the actual commands that will be executed on the runner in this job. There can be one or many commands in each job. You can use predefined environmental variables in commands, as you see in the example where the *CI_PROJECT_DIR* variable is used. The variable results in the location where the repository is cloned on the runner. For more variables, consult the documentation.

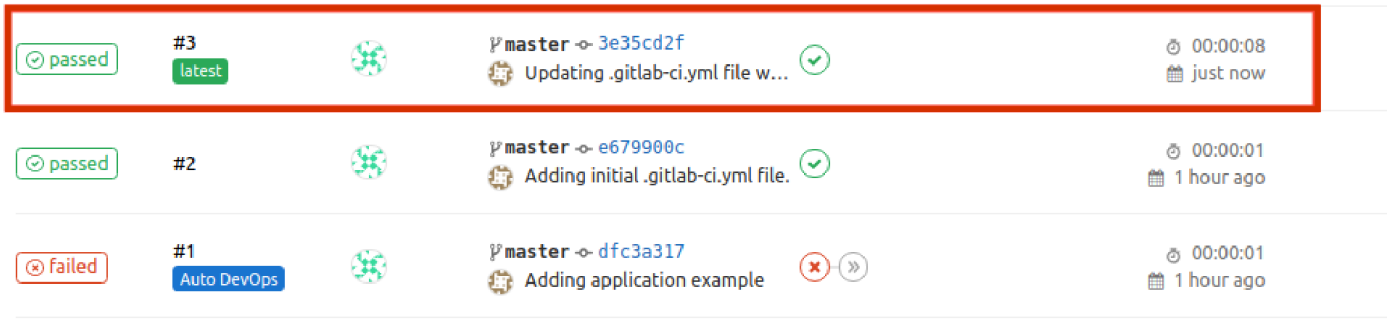
In this example, the command first enters the *app* folder in the repository and then builds the *app* container.

Step 2: Commit and push the changes to the repository.

```
student@student-workstation:~/working_directory$ git commit -am "Updating .gitlab-ci.yml file with the build
[master 3e35cd2] Updating .gitlab-ci.yml file with the build stage
 1 file changed, 6 insertions(+), 2 deletions(-)
student@student-workstation:~/working_directory$ git push
Username for 'http://dev.gitlab.local': root
```

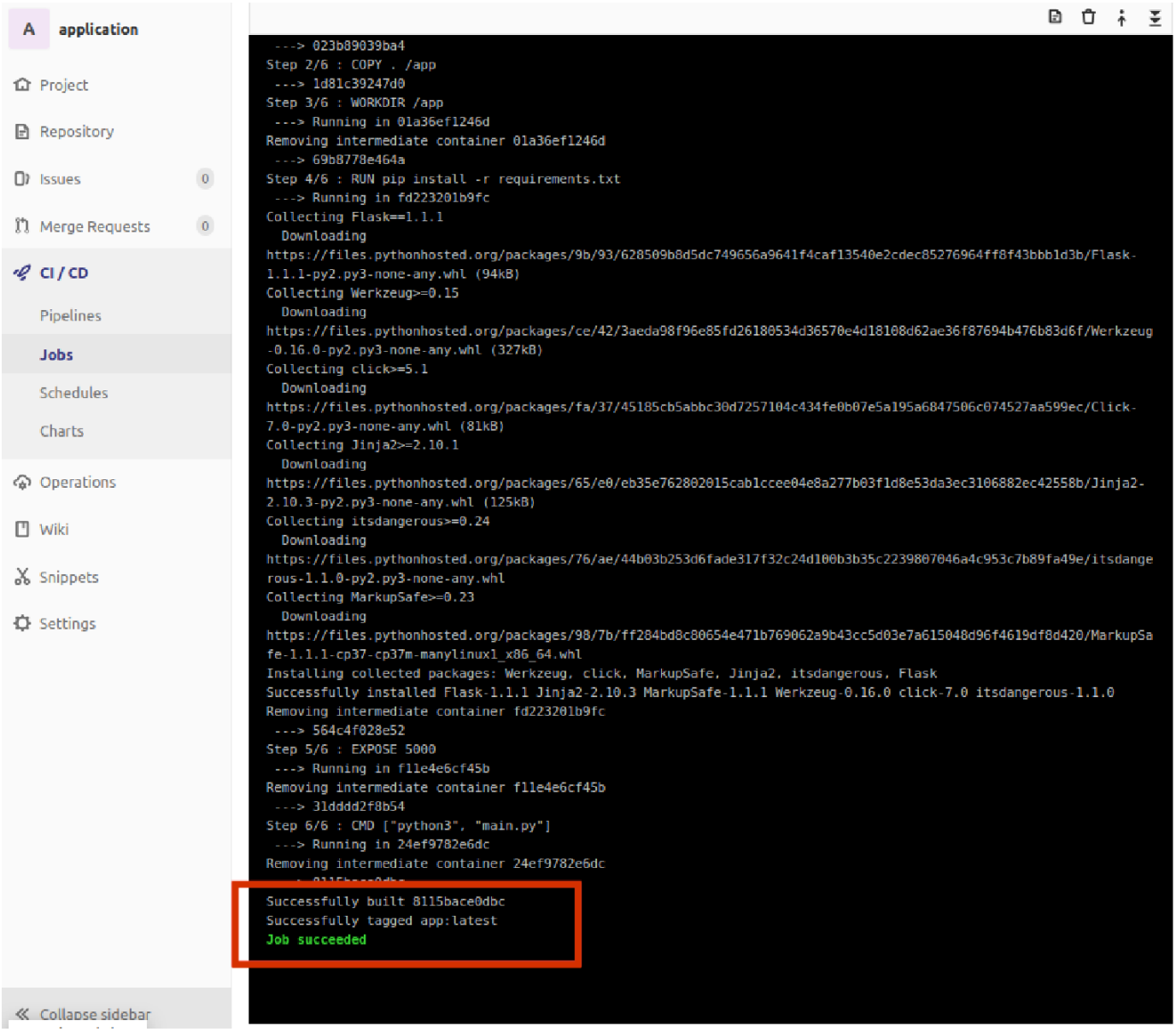
```
Password for 'http://root@dev.gitlab.local': 1234QWer
Counting objects: 3, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 406 bytes | 406.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To http://dev.gitlab.local/root/application.git
   e679900..3e35cd2  master -> master
```

Step 3: In the GitLab web user interface, navigate to **CI/CD > Pipelines** to check the status of the pipeline execution.



You can see that the pipeline was executed successfully.

Step 4: Open the job details and check the job output.



As you can see from the output, the job was executed successfully.

Step 5: Manually start the *app* container. Use the **docker run -it -p 5000:5000 app** command.

```
student@student-workstation:~/working_directory$ docker run -it -p 5000:5000 app
 * Serving Flask app "main" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 674-030-449
```

You can see that the container was started correctly. You can stop the container for now by pressing **Ctrl-C**.

Task 4: Implement Unit Test Stage

In this procedure, you will implement simple unit tests. Unit tests will test the basic application behavior. After you implement all the unit tests, add an extra stage to your CI/CD pipeline, which executes the unit tests automatically every time you push changes to the application.

Activity

Step 1: In the *app* folder, create a new folder with the name *tests*. In this folder, add the file with the name *app_tests.py*.

Step 2: In the *app_tests.py* file, create a skeleton for the tests. Create a new class called *AppTest*, which inherits the *unittest.TestCase* class. Add the *setUp* method, in which you store the URL for your application. Use *http://localhost:5000* for the URL. Finally, add the code that runs the *unittest.main()* function when you execute the file.

```
import unittest


class AppTest(unittest.TestCase):

    def setUp(self):
        self.url = 'http://localhost:5000'


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

The *unittest* Python module is usually used for unit testing. To define tests, you can create a new class, which inherits *unittest.TestCase* class. The *unittest.TestCase* class has special methods that are called before tests are executed. The *setUp()* method is called before each test is executed. You will add the URL for your application to the *setUp* method. The tests are then defined with the methods, where each method needs to start with the *test_* keyword. The module executes the tests when you call the *main()* function.

Step 3: Add a simple test to the *AppTest* class. The test should send the GET request to the application. After a response is received, you should check that status code is 200 and that you can see the "*Welcome to Cisco DevNet*." text in the output. Name your test *test_welcome*.

```
import unittest
import requests

class AppTest(unittest.TestCase):

    def setUp(self):
        self.url = 'http://localhost:5000'

    def test_welcome(self):
        response = requests.get(self.url)
        status_code = response.status_code
        content = response.content.decode('ascii')

        self.assertEqual(status_code, 200)
        self.assertIn('Welcome to Cisco DevNet.', content)

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

To generate the HTTP request, you can use the *requests* Python module. When response is received, you can extract the status code and the content. Once you have both values, you can use the **unittest** methods **assertEqual** and **assertIn** to compare the values. The **assertEqual** performs the *==* operation between values, while **assertIn** method performs the *in* operation to check if first value is found in the second value.

Step 4: Start the app container with the **docker run -it -p 5000:5000 app** command. In the second terminal, use the **python app_tests.py** command in the *app* folder to run the test.

```
student@student-workstation:~/working_directory/app/tests$ python app_tests.py
.
----------------------------------------------------------------------
Ran 1 test in 0.007s

OK
```

You can see that the test was executed. The executed tests are marked with dots. If all tests are successful, you can see *OK* at the end.

Step 5: Change the text '*Welcome to Cisco DevNet.*' to '*Welcome home.*', to see the output when the test is failing. Run the test script again.

```
<... output omitted ...>
    def test_welcome(self):
        response = requests.get(self.url)
        status_code = response.status_code
        content = response.content.decode('ascii')
```

```
        self.assertEqual(status_code, 200)
        self.assertIn('Welcome home.', content)
<... output omitted ...>
student@student-workstation:~/working_directory/app/tests$ python app_tests.py
F
======================================================================
FAIL: test_welcome (__main__.AppTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "app_tests.py", line 16, in test_welcome
    self.assertIn('Welcome home.', content)
AssertionError: 'Welcome home.' not found in 'Welcome to Cisco DevNet.<br>IP address of the server is 172.18.

----------------------------------------------------------------------
Ran 1 test in 0.009s

FAILED (failures=1)
```

When the result of the condition is *False*, the script throws the *AssertionError* exception and marks the test as failed.

Step 6: Change the text back to '*Welcome to Cisco DevNet*.'.

Step 7: Create a new test called *test_welcome_negative*. The test will be negative, which should test that the '*Welcome home*.' string is not found in the output.

```
import unittest
import requests

class AppTest(unittest.TestCase):

    def setUp(self):
        self.url = 'http://localhost:5000'

    def test_welcome(self):
        response = requests.get(self.url)
        status_code = response.status_code
        content = response.content.decode('ascii')

        self.assertEqual(status_code, 200)
        self.assertIn('Welcome to Cisco DevNet.', content)

    def test_welcome_negative(self):
        response = requests.get(self.url)
        status_code = response.status_code
        content = response.content.decode('ascii')

        self.assertEqual(status_code, 200)
        self.assertNotIn('Welcome home.', content)

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

To test that the particular string is not found in the content, you can use **assertNotIn** method.

Step 8: Add the third test named **test_ip**, which should test that the IP address of the server is found in the output.

```
import unittest
import requests

class AppTest(unittest.TestCase):

    def setUp(self):
        self.url = 'http://localhost:5000'

    def test_welcome(self):
        response = requests.get(self.url)
        status_code = response.status_code
        content = response.content.decode('ascii')

        self.assertEqual(status_code, 200)
        self.assertIn('Welcome to Cisco DevNet.', content)

    def test_welcome_negative(self):
        response = requests.get(self.url)
        status_code = response.status_code
        content = response.content.decode('ascii')

        self.assertEqual(status_code, 200)
        self.assertNotIn('Welcome home.', content)

    def test_ip(self):
        response = requests.get(self.url)
        status_code = response.status_code
        content = response.content.decode('ascii')
```

```
        self.assertEqual(status_code, 200)
        ip_regex = 'IP address of the server is ([0-9]{1,3}\.){3}[0-9]{1,3}.'
        self.assertRegex(content, ip_regex)

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

Because the IP address can be different, you need to use the regular expression to check the presence of the IP address in the content. The example shows the most basic regular expression to match the IP address. This regular expression is not 100 percent accurate, but it is enough for this example. To use the regular expression comparison, use the **assertRegex** method.

Step 9: Run the test script again and check that all three tests pass the execution.

```
student@student-workstation:~/working_directory/app/tests$ python app_tests.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.016s

OK
```

You should see that all three tests pass the execution. Now you can run these tests as part of the CI/CD pipeline.

Step 10: Stop the running *app* container by using **Ctrl-C.**

Step 11: Create a script that will be used to start the *app* container in the background, execute the tests, and stop the container. Add the script to the *app/tests* folder. Call your script **run_tests.sh**.

```
#!/bin/bash

DOCKER_ID=`docker run -d -p 5000:5000 app`
sleep 3

python app_tests.py
EXIT_CODE=$?

docker kill $DOCKER_ID
docker rm $DOCKER_ID
exit $EXIT_CODE
```

The script executes the **docker run** command and stores the container ID in the variable by using backticks. The script waits for 3 seconds, so that application is running, and executes the tests. After the execution, the script stores the exit code. The exit code value of the last command that was executed on the server is $?. It is important to pass the correct exit code when you finish the script, so that GitLab will know if it needs to stop executing the pipeline or not. GitLab stops executing the pipeline when the exit code is nonzero. Before the script is finished, the script stops and removes the running container.

Step 12: Execute the script in the *app/tests* folder with the **/bin/bash run_tests.sh** command.

```
student@student-workstation:~/working_directory/app/tests$ /bin/bash run_tests.sh
...
----------------------------------------------------------------------
Ran 3 tests in 0.015s

OK
0d8dd1c9284b2fade921d134afbb23f8f616e91eb6afa778b1651cd0c923bdd2
0d8dd1c9284b2fade921d134afbb23f8f616e91eb6afa778b1651cd0c923bdd2
```

You should see that the script executes the tests. The last two hash values mean that the container was stopped and removed.

Step 13: Now update the *.gitlab-ci.yml* file. Add a new stage, called *unittest*, to the *.gitlab-ci.yml* file. Define the job with the name *unittest_application_code*, which should execute the tests.

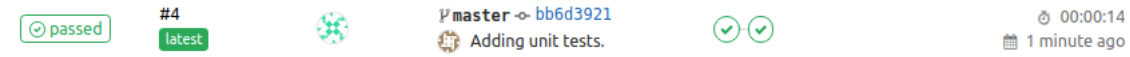
```
stages:
  - build
  - unittest

build:
  stage: build
  script: cd $CI_PROJECT_DIR/app && docker build -t app .

unittest_application_code:
  stage: unittest
  script: cd $CI_PROJECT_DIR/app/tests && /bin/bash run_tests.sh
```

Step 14: Commit and push the changes to GitLab.

```
student@student-workstation:~/working_directory/app$ git add tests/
student@student-workstation:~/working_directory/app$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   tests/app_tests.py
        new file:   tests/run_tests.sh

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
```

```
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   ../.gitlab-ci.yml

student@student-workstation:~/working_directory/app$ git commit -am "Adding unit tests."
[master bb6d392] Adding unit tests.
 3 files changed, 53 insertions(+), 1 deletion(-)
 create mode 100644 app/tests/app_tests.py
 create mode 100644 app/tests/run_tests.sh
student@student-workstation:~/working_directory/app$ git push
Username for 'http://dev.gitlab.local': root
Password for 'http://root@dev.gitlab.local': 1234QWer
Counting objects: 7, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (7/7), done.
Writing objects: 100% (7/7), 1.03 KiB | 1.03 MiB/s, done.
Total 7 (delta 1), reused 0 (delta 0)
To http://dev.gitlab.local/root/application.git
   ad9c714..bb6d392  master -> master
```

Step 15: Check the pipelines on GitLab by navigating to the **CI/CD > Pipelines**. Open the *unittest* stage.




```
Running with gitlab-runner 12.4.1 (05161b14)
  on student-workstation oQ8CNNxP
Using Shell executor...
Running on student-workstation...
Fetching changes with git depth set to 50...
Reinitialized existing Git repository in /home/gitlab-runner/builds/oQ8CNNxP/0/root/application/.git/
Checking out bb6d3921 as master...
Skipping Git submodules setup
$ cd $CI_PROJECT_DIR/app/tests && /bin/bash run_tests.sh
...
----------------------------------------------------------------------
Ran 3 tests in 0.020s

OK
558a4562d386c65c71369b3804a13d93f949b446120825bd81c80dd0d586c12b
558a4562d386c65c71369b3804a13d93f949b446120825bd81c80dd0d586c12b
Job succeeded
```

You should see that the status of the pipeline is *passed*. In the job output, you can see that the tests were executed correctly.

Task 5: Implement Packaging Stage

In this procedure, you will define the packaging stage. In this stage, you will prepare production ready Docker containers. You will push these containers to Docker registry, which will be hosted on GitLab.

Activity

Step 1: Add a new stage called *systembuild* to the *.gitlab-ci.yml* file.

```
stages:
  - build
  - unittest
  - systembuild
```

Step 2: Add a new job to the *.gitlab-ci.yml* file. Use the name *build_app* and add the job to the *systembuild* stage. In this job, you should rebuild the *app* container and push the container to the local Docker registry, which is located on the GitLab server. To rebuild and push the Docker image to the registry, use the following commands:

```
docker login http://dev.gitlab.local:5005 -u root -p 1234QWer
docker build -t dev.gitlab.local:5005/root/application/app .
docker push dev.gitlab.local:5005/root/application/app
```

```
stages:
  - build
  - unittest
  - systembuild

build:
  stage: build
  script: cd $CI_PROJECT_DIR/app && docker build -t app .

unittest_application_code:
  stage: unittest
  script: cd $CI_PROJECT_DIR/app/tests && /bin/bash run_tests.sh

build_app:
  stage: systembuild
  script:
    - cd $CI_PROJECT_DIR/app
    - docker login http://dev.gitlab.local:5005 -u root -p 1234QWer
```

```
    - docker build -t dev.gitlab.local:5005/root/application/app .
    - docker push dev.gitlab.local:5005/root/application/app
```

The docker commands will first log in to the registry, then build the image, and finally push the image to the registry. You need to use the complete path for the Docker image name.

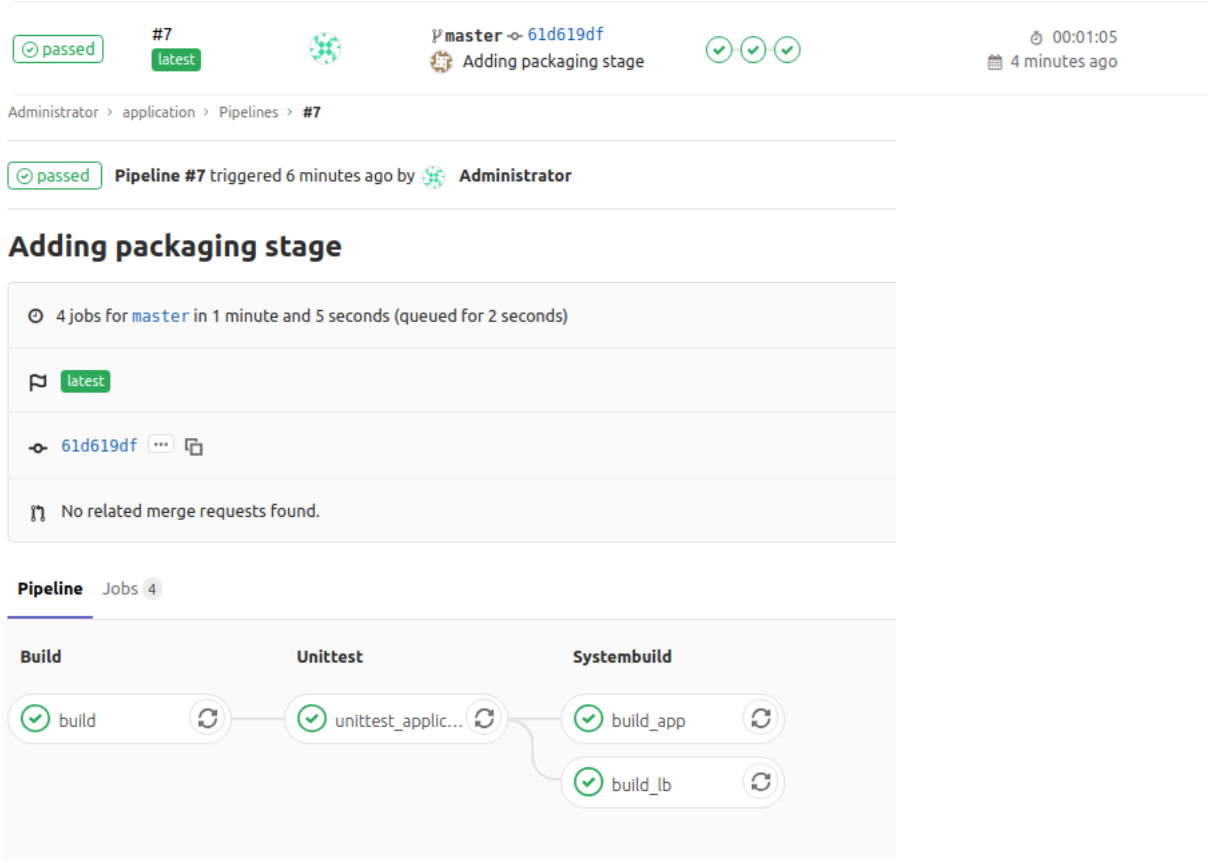
Step 3: Create a similar job for building the *lb* container. Use the name *build_lb*.

```
stages:
  - build
  - unittest
  - systembuild

build:
  stage: build
  script: cd $CI_PROJECT_DIR/app && docker build -t app .

unittest_application_code:
  stage: unittest
  script: cd $CI_PROJECT_DIR/app/tests && /bin/bash run_tests.sh

build_app:
  stage: systembuild
  script:
    - cd $CI_PROJECT_DIR/app
    - docker login http://dev.gitlab.local:5005 -u root -p 1234QWer
    - docker build dev.gitlab.local:5005/root/application/app
    - docker push dev.gitlab.local:5005/root/application/app

build_lb:
  stage: systembuild
  script:
    - cd $CI_PROJECT_DIR/lb
    - docker login http://dev.gitlab.local:5005 -u root -p 1234QWer
    - docker build -t dev.gitlab.local:5005/root/application/lb .
    - docker push dev.gitlab.local:5005/root/application/lb
```

Step 4: Commit the changes and push the changes to GitLab.

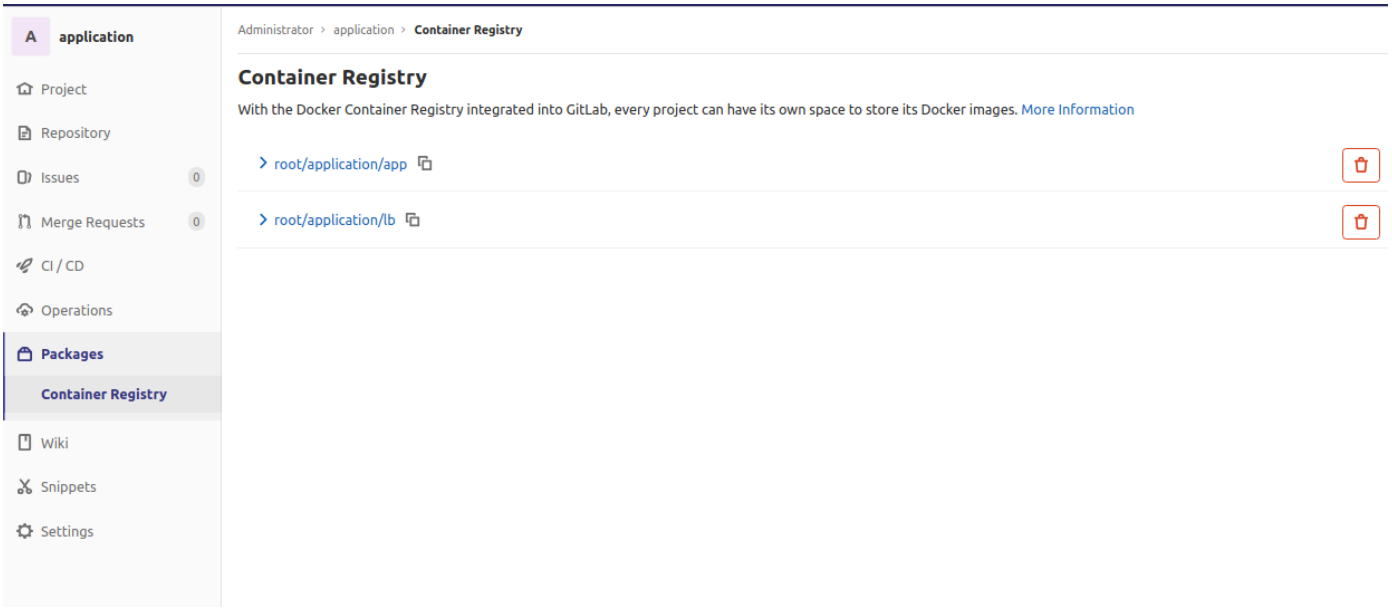
```
student@student-workstation:~/working_directory$ git commit -am "Adding packaging stage"
[master ece319c] Adding packaging stage
 1 file changed, 19 insertions(+), 1 deletion(-)
student@student-workstation:~/working_directory$ git push
Username for 'http://dev.gitlab.local': root
Password for 'http://root@dev.gitlab.local': 1234QWer
Counting objects: 3, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 547 bytes | 547.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To http://dev.gitlab.local/root/application.git
   bb6d392..ece319c  master -> master
```

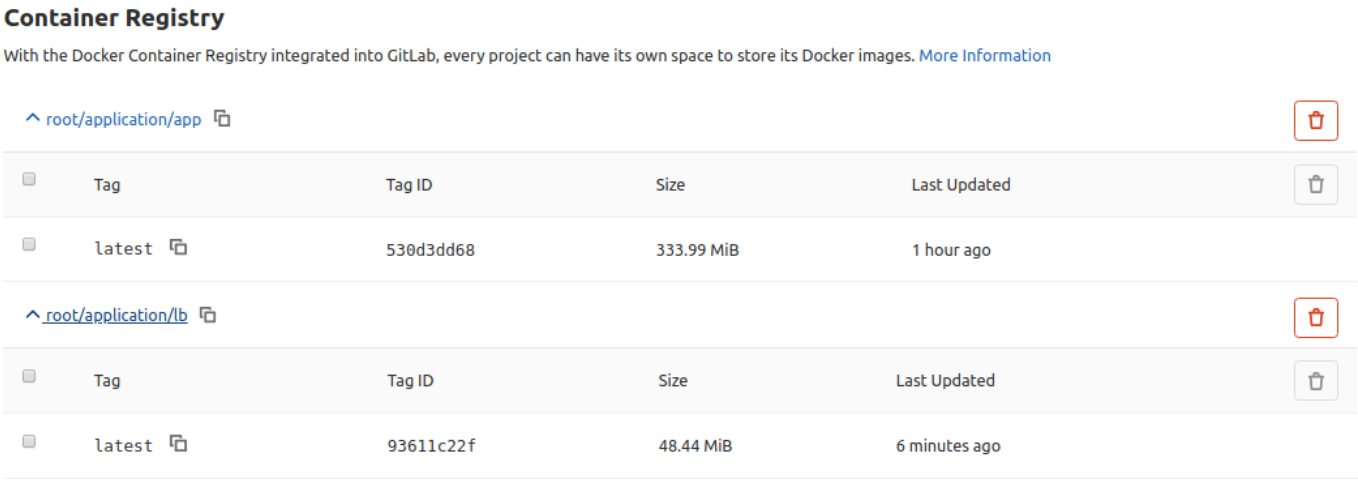
Step 5: Check the pipelines on GitLab. Navigate to **CI/CD > Pipelines**. Click the **passed** button.



You can see that the pipeline status is passed. If you click the **passed** button, you can see the pipeline workflow. You can see that the jobs in the *systembuild* stage were executed in parallel.

Step 6: Open the **Packages > Container** Registry and expand both containers.





You should see that both containers are in the Docker registry on GitLab. These containers can now be used from any host that has access to the registry. For example, you can use a GitLab runner to build the production ready containers, and then you can deploy the containers to the production servers from the GitLab registry.

Task 6: Implement System Test Stage

In this procedure, you will implement system tests. System tests check application behavior when using all application components. In this example, you will test the application when deployed with a load balancer and two application containers. You will add system tests as part of the CI/CD pipeline.

Activity

Step 1: Create a new folder with the name *tests* in the top-level directory. Add the file **system_tests.py** to the folder.

Step 2: Create a skeleton for your tests. Use a similar skeleton to the unit tests. This time use the *http://localhost:8080* URL in the *setUp* method.

```
import unittest

class AppTest(unittest.TestCase);

    def setUp(self):
        self.url = 'http://localhost:8080'

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

Step 3: Add the first test to the file. Use the name *test_welcome*.

Perform the following basic tests:

Make sure that the status code of the response is 200.
Make sure that you see the "Welcome to Cisco DevNet." text in the output.
Make sure that you do not see the "Welcome home." text in the output.
Make sure that you see the IP address in the output.

```
import unittest
import requests

class AppTest(unittest.TestCase):

    def setUp(self):
        self.url = 'http://localhost:8080'

    def test_welcome(self):
        response = requests.get(self.url)
        status_code = response.status_code
        content = response.content.decode('ascii')

        self.assertEqual(status_code, 200)
        self.assertIn('Welcome to Cisco DevNet.', content)
        self.assertNotIn('Welcome home.', content)
        ip_regex = 'IP address of the server is ([0-9]{1,3}\.){3}[0-9]{1,3}.'
        self.assertRegex(content, ip_regex)


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

The following test executes similar checks as unit tests.

Step 4: Add another test named *test_nginx*. In this test, you should check that the load balancer adds the *Server* header to the response. The header should have the value *nginx*.

```
import unittest
import requests

class AppTest(unittest.TestCase):

    def setUp(self):
        self.url = 'http://localhost:8080'

    def test_welcome(self):
        response = requests.get(self.url)
        status_code = response.status_code
        content = response.content.decode('ascii')

        self.assertEqual(status_code, 200)
        self.assertIn('Welcome to Cisco DevNet.', content)
        self.assertNotIn('Welcome home.', content)
        ip_regex = 'IP address of the server is ([0-9]{1,3}\.){3}[0-9]{1,3}.'
        self.assertRegex(content, ip_regex)

    def test_nginx(self):
        response = requests.get(self.url)
        status_code = response.status_code
        headers = response.headers
        server_header = headers.get('Server')

        self.assertEqual(status_code, 200)
        self.assertIsNot(server_header, None)
        self.assertIn('nginx', server_header)


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

As you can see, the test checks that status code 200 is returned. Also, the test checks that there is the *Server* header in the response and that the value of the *Server* header is *nginx*.

Step 5: Add the third test that checks the load balancing. Name your test *test_lb*. You should make two requests and you need to make sure that the IP addresses in two responses are not equal.

```
import unittest
import requests
import re

class AppTest(unittest.TestCase):

    def setUp(self):
        self.url = 'http://localhost:8080'

    def test_welcome(self):
        response = requests.get(self.url)
        status_code = response.status_code
        content = response.content.decode('ascii')

        self.assertEqual(status_code, 200)
        self.assertIn('Welcome to Cisco DevNet.', content)
        self.assertNotIn('Welcome home.', content)
```

```
        ip_regex = 'IP address of the server is ([0-9]{1,3}\.){3}[0-9]{1,3}.'
        self.assertRegex(content, ip_regex)

    def test_nginx(self):
        response = requests.get(self.url)
        status_code = response.status_code
        headers = response.headers
        server_header = headers.get('Server')

        self.assertEqual(status_code, 200)
        self.assertIsNot(server_header, None)
        self.assertIn('nginx', server_header)

    def test_lb(self):
        response1 = requests.get(self.url)
        response2 = requests.get(self.url)

        content1 = response1.content.decode('ascii')
        content2 = response2.content.decode('ascii')

        ip_regex = '([0-9]{1,3}\.){3}[0-9]{1,3}.'

        ip_search1 = re.search(ip_regex, content1)
        ip_search2 = re.search(ip_regex, content2)

        self.assertIsNot(ip_search1, None)
        self.assertIsNot(ip_search2, None)
        self.assertNotEqual(ip_search1.group(), ip_search2.group())


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

You should see that the code makes two requests. Then the IP address is searched in each request. Finally, the code checks that IP addresses are present and that IP addresses are not equal.

Step 6: Add a test execution script to the tests folder. Name your script **run_system_tests.sh**. The script should check if the Docker network *appnet* exists. If not, the script should create the Docker network with the subnet 172.20.0.0/24 and gateway 172.20.0.1. Then the script should start 1 *lb* container in the background. The IP address of the container should be 172.20.0.10 and the exposed TCP port should be 8080. Then the script should start two app containers with the IP addresses 172.20.0.100 and 172.20.0.101. Use the container that you have built in the *systembuild* stage. After starting the containers, the script should execute the system tests. At the end, the script should stop and remove the containers. The exit code of the script should be the same as the exit code of the system tests.

```
#!/bin/bash

docker network inspect appnet

if [[ $? -ne 0 ]]; then
    docker network create --subnet=172.20.0.0/24 --gateway=172.20.0.1 appnet
fi

DOCKER_LB_ID=`docker run --net appnet --ip 172.20.0.10 -p 8080:8080 -itd dev.gitlab.local:5005/root/applicati
DOCKER_APP1_ID=`docker run --net appnet --ip 172.20.0.100 -itd dev.gitlab.local:5005/root/application/app`
DOCKER_APP2_ID=`docker run --net appnet --ip 172.20.0.101 -itd dev.gitlab.local:5005/root/application/app`
sleep 5

python tests/system_tests.py
EXIT_CODE=$?

docker stop $DOCKER_LB_ID
docker rm $DOCKER_LB_ID
docker stop $DOCKER_APP1_ID
docker rm $DOCKER_APP1_ID
docker stop $DOCKER_APP2_ID
docker rm $DOCKER_APP2_ID
exit $EXIT_CODE
```

Step 7: Add a new stage to the *.gitlab-ci.yml* file. The name of the stage should be *systemtest*. Add the new job called *system_test*, use the stage *systemtest,* and execute the script *run_system_tests.sh*.

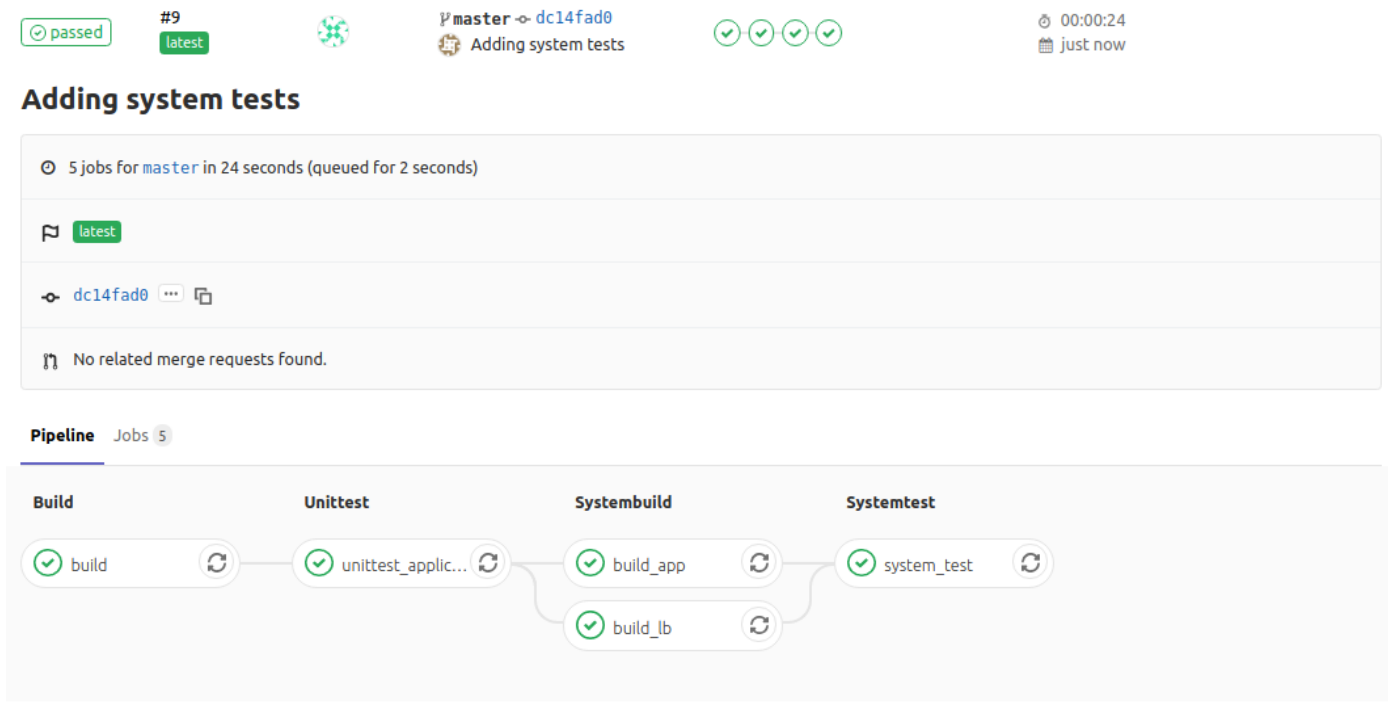
```
stages:
  - build
  - unittest
  - systembuild
  - systemtest

build:
  stage: build
  script: cd $CI_PROJECT_DIR/app && docker build -t app .

unittest_application_code:
  stage: unittest
  script: cd $CI_PROJECT_DIR/app/tests && /bin/bash run_tests.sh

build_app:
```

```
  stage: systembuild
  script:
    - cd $CI_PROJECT_DIR/app
    - docker login http://dev.gitlab.local:5005 -u root -p 1234QWer
    - docker build -t dev.gitlab.local:5005/root/application/app .
    - docker push dev.gitlab.local:5005/root/application/app

build_lb:
  stage: systembuild
  script:
    - cd $CI_PROJECT_DIR/lb
    - docker login http://dev.gitlab.local:5005 -u root -p 1234QWer
    - docker build -t dev.gitlab.local:5005/root/application/lb .
    - docker push dev.gitlab.local:5005/root/application/lb

system_test:
  stage: systemtest
  script: /bin/bash tests/run_system_tests.sh
```

Step 8: Commit the changes and push the changes to GitLab.

```
student@student-workstation:~/working_directory$ git add tests
student@student-workstation:~/working_directory$ git commit -am "Adding system tests"
[master 4cf9f13] Adding system tests
 3 files changed, 79 insertions(+)
 create mode 100644 tests/run_system_tests.sh
 create mode 100644 tests/system_tests.py
student@student-workstation:~/working_directory$ git push
Username for 'http://dev.gitlab.local': root
Password for 'http://root@dev.gitlab.local': 1234QWer
Counting objects: 6, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 1.23 KiB | 1.23 MiB/s, done.
Total 6 (delta 1), reused 0 (delta 0)
To http://dev.gitlab.local/root/application.git
   61d619d..4cf9f13  master -> master
```

Step 9: Check the pipelines in GitLab. Click the **passed** button.



You should see the complete workflow of your CI/CD pipeline.

Task 7: Test CI/CD Pipeline

In this procedure, you will intentionally add two errors to your application. You will observe if your tests catch those errors and you will fix these errors to restore the application.

Activity

Step 1: Add an error into the *app/main.py* file. Comment out the line where you add the IP address to the output.

```
from flask import Flask
import socket
```

```
ip = socket.gethostbyname(socket.gethostname())

app = Flask(__name__)

@app.route('/')
def home():
    out = (
        f'Welcome to Cisco DevNet.<br>'
        # f'IP address of the server is {ip}.<br><br>'
    )
    return out

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0')
```

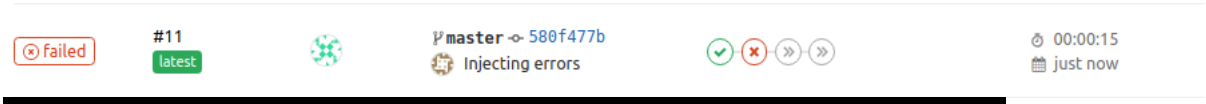
Step 2: Add an error into the *lb/nginx.conf* file. Comment out the second upstream server.

```
events {}
http {

  upstream myapp {
    server 172.20.0.100:5000;
    # server 172.20.0.101:5000;
  }

  server {
    listen 8080;
    server_name localhost;

    location / {
      proxy_pass http://myapp;
      proxy_set_header Host $host;
    }
  }

}
```

Step 3: Commit the changes and push the changes to GitLab.

```
student@student-workstation:~/working_directory$ git commit -am "Injecting errors"
[master 580f477] Injecting errors
 2 files changed, 2 insertions(+), 2 deletions(-)
student@student-workstation:~/working_directory$ git push
Username for 'http://dev.gitlab.local': root
Password for 'http://root@dev.gitlab.local': 1234QWer
Counting objects: 6, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 543 bytes | 543.00 KiB/s, done.
Total 6 (delta 3), reused 0 (delta 0)
To http://dev.gitlab.local/root/application.git
   5d937f8..580f477  master -> master
```

Step 4: Check the pipelines in GitLab.



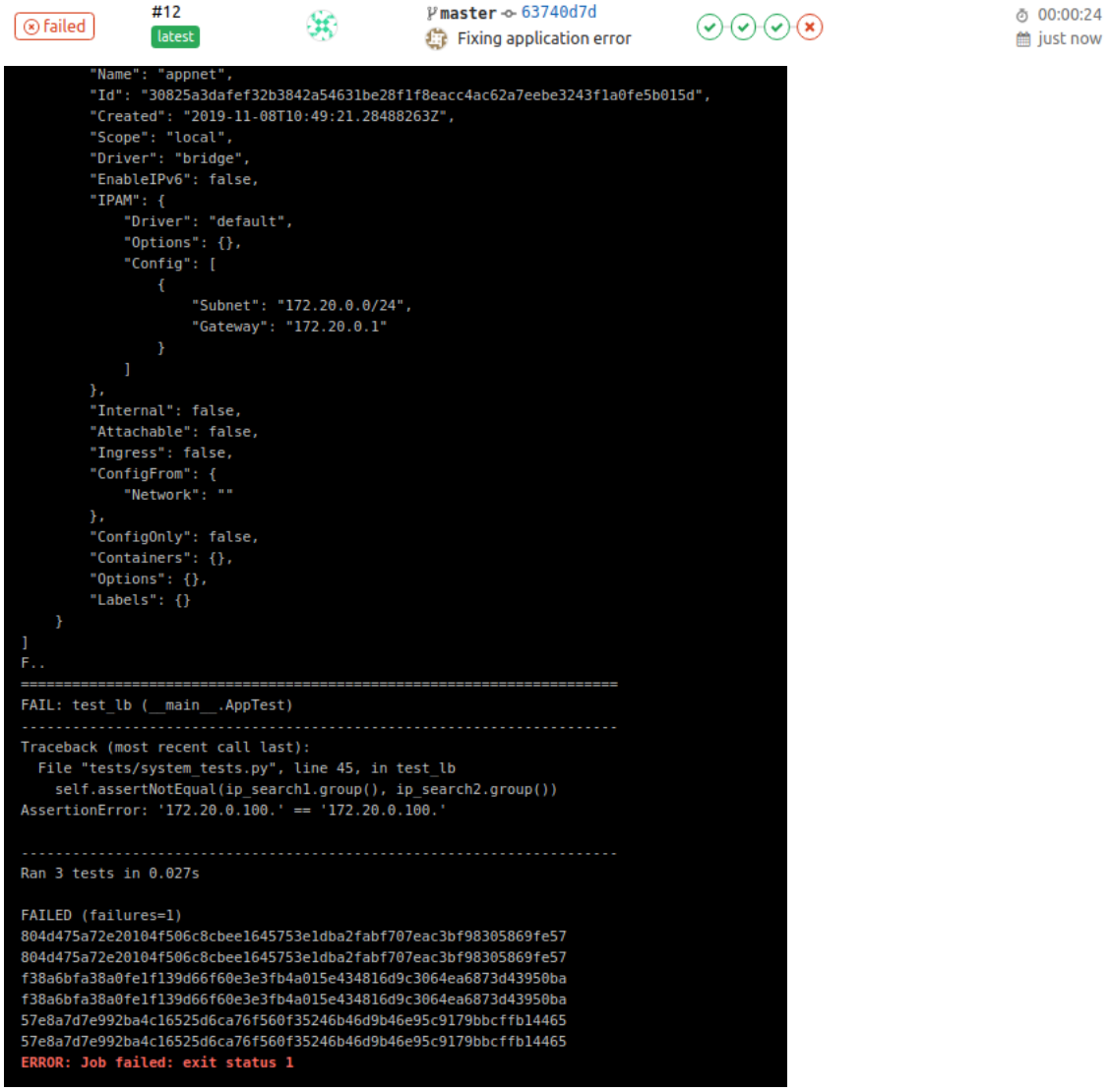

```
Running with gitlab-runner 12.4.1 (05161b14)
  on student-workstation oQ8CNNxP
Using Shell executor...
Running on student-workstation...
Fetching changes with git depth set to 50...
Reinitialized existing Git repository in /home/gitlab-runner/builds/oQ8CNNxP/0/root/application/.git/
Checking out 580f477b as master...
Skipping Git submodules setup
$ cd $CI_PROJECT_DIR/app/tests && /bin/bash run_tests.sh
F..
======================================================================
FAIL: test_ip (__main__.AppTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "app_tests.py", line 32, in test_ip
    self.assertRegex(content, ip_regex)
AssertionError: Regex didn't match: 'IP address of the server is ([0-9]{1,3}\\.){3}[0-9]{1,3}.' not found in
'Welcome to Cisco DevNet.<br>'

----------------------------------------------------------------------
Ran 3 tests in 0.018s

FAILED (failures=1)
31a5e85b0fd9fb9a4c38cfc23fa0bd8835954a9924c2ec1cbf85397a7cbab042
31a5e85b0fd9fb9a4c38cfc23fa0bd8835954a9924c2ec1cbf85397a7cbab042
ERROR: Job failed: exit status 1
```

You can see that the pipeline status is failed. The failure was in the *unittest* stage. GitLab stopped execution when nonzero exit code was seen.

Step 5: Uncomment the commented line in the *app/main.py*. Commit the changes and push the changes to GitLab.

```
from flask import Flask
import socket

ip = socket.gethostbyname(socket.gethostname())

app = Flask(__name__)

@app.route('/')
def home():
    out = (
        f'Welcome to Cisco DevNet.<br>'
        f'IP address of the server is {ip}.<br><br>'
    )
    return out

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0')
student@student-workstation:~/working_directory$ git commit -am "Fixing application error"
[master 63740d7] Fixing application error
 1 file changed, 1 insertion(+), 1 deletion(-)
student@student-workstation:~/working_directory$ git push
Username for 'http://dev.gitlab.local': root
Password for 'http://root@dev.gitlab.local': 1234QWer
Counting objects: 4, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 423 bytes | 423.00 KiB/s, done.
Total 4 (delta 2), reused 0 (delta 0)
To http://dev.gitlab.local/root/application.git
   580f477..63740d7  master -> master
```

Step 6: Check the pipelines in GitLab.



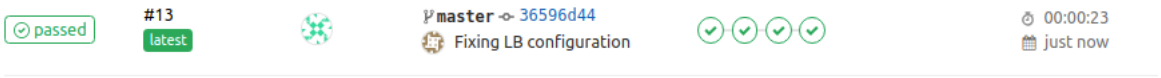
You can see that this time there was a failure at the last stage of the pipeline. You can see that there was a failure in one of the tests.

Step 7: Uncomment the commented line in the *lb/nginx.conf*. Commit the changes and push the changes to GitLab.

```
events {}
http {

  upstream myapp {
    server 172.20.0.100:5000;
    server 172.20.0.101:5000;
  }

  server {
    listen 8080;
    server_name localhost;

    location / {
      proxy_pass http://myapp;
      proxy_set_header Host $host;
    }
  }

}
student@student-workstation:~/working_directory$ git commit -am "Fixing LB configuration"
[master 36596d4] Fixing LB configuration
 1 file changed, 1 insertion(+), 1 deletion(-)
student@student-workstation:~/working_directory$ git push
Username for 'http://dev.gitlab.local': root
Password for 'http://root@dev.gitlab.local': 1234QWer
Counting objects: 4, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 449 bytes | 449.00 KiB/s, done.
Total 4 (delta 1), reused 0 (delta 0)
```

```
To http://dev.gitlab.local/root/application.git
   63740d7..36596d4  master -> master
```

Step 8: Check the pipelines in GitLab.



You can see that the pipeline was executed correctly this time, which means that you have successfully fixed the issues in your application.