Week 4 lab tutorial – help

Wednesday, 16 August 2023 12:33 PM

Fundamental goals
- Learn about CI/CD - catch problems early
- Learn about Docker

Activity 1

1. Fork and Clone the Repository:	Navigate to the repository on your workspace (https://gitlab.cecs.anu.edu.au/comp2120/2023/comp2120-tut4-ci) Click on the "Fork" button to create a copy of the repository in your Gitlab account. Use your Gitlab university credentials to login. Once forked, clone the repository to your local machine using the following command: `git clone <URL_of_your_forked_repository>`
2. Understand the Repository:	The repository is a solution to a problem from Codeforces. It uses JUnit 4 for testing. The build system is Gradle. (Note: Maven is another popular build system, but the commands for building and testing would be similar.)
3. Set Up the CI Pipeline:	Navigate to the root of the project. Create the .gitlab-ci.yml file. Write the CI pipeline in the YAML format as described below: ``` stages: - build - test gradle-build: stage: build tags: - comp2100 script: - ./gradlew assemble artifacts: paths: - build/ gradle-test: stage: test tags: - comp2100 script: - ./gradlew test artifacts: reports: junit: build/test-results/test/TEST-*.xml ``` Explanation: • The stages keyword defines the order in which jobs are executed. Here, we have two stages: build and test. • gradle-build is the job that compiles the source code into application bytecode using the gradle assemble command. • gradle-test is the job that runs all the test cases for the project using the gradle test command. • The artifacts keyword specifies the list of files and directories that are attached to the job when it succeeds. In the gradle-test job, we're specifying that the JUnit test results should be saved as an artifact. • The tags references the runner that will be used for the job. Here, we use 'comp2100' runner that is available on Gitlab server.
4. Commit and Push:	After writing the CI pipeline, save the .gitlab-ci.yml file. Commit the changes: ``` git add .gitlab-ci.yml git commit -m "Add CI pipeline for building and testing" ``` Push the changes to your forked repository:

	`git push origin master`
5. Verify the Pipeline:	Navigate to your GitLab repository. Go to CI/CD -> Pipelines. Click on the job number to see the pipeline you've just created. It should match the described structure.

Potential issues: gradle-build fails because Docker container (in which CI job is running) doesn't recognise `gradle` command – can resolve by the fact that the repo has a Gradle wrapper.

How to setup annotated test results in the browser

1. Generate JUnit Test Reports:	Ensure your build.gradle file is configured to produce JUnit XML reports.

2. Update .gitlab-ci.yml:	You've already set up the artifacts section in your .gitlab-ci.yml to collect JUnit test results:
	``
	artifacts:
	  reports:
	  junit: build/test-results/test/TEST-*.xml
	``
	
	This configuration tells GitLab CI/CD to collect the JUnit XML reports as artifacts and use them for test report visualization.

3. View Annotated Test Results:	Once the pipeline runs: • Navigate to your GitLab project. • Go to CI/CD > Pipelines. • Click on the pipeline that you want to view. • In the pipeline details, you'll see a Test Report tab (next to Jobs). Click on it. • Here, you'll see the annotated test results. Failed tests will be highlighted, and you can click on each test to see more details.

Activity 2

Aim: make CI/CD pipeline only run for pull requests (merge requests in GitLab terminology)
Method: use the rules keyword in the .gitlab-ci.yml file. The rules keyword allows you to define conditions for when jobs should run.

1. Modify your .gitlab-ci.yml file to run the gradle-test job only during merge requests:	```
	stages:
	  - build
	  - test
	
	gradle-build:
	  stage: build
	  tags:
	  - comp2100
	  script:
	  - ./gradlew assemble
	  artifacts:
	  paths:
	  - build/
	
	gradle-test:
	  stage: test
	  tags:
	  - comp2100
	  script:
	  - ./gradlew test
	  artifacts:
	  reports:
	  junit: build/test-results/test/TEST-*.xml
	  rules:
	  - if: '$CI_PIPELINE_SOURCE == "merge_request_event"'
	``

2. Further refine pipeline (optional)	For instance, to run a proselint job only when documentation changes: ``` proselint:

For example, to skip extensive tests on documentation changes, you can add more rules and use the changes keyword to specify paths or files to check for changes.	stage: test script: - proselint docs/ rules: - changes: - docs/**/* ``` This proselint job will only run if there are changes in the docs/ directory.

Expected result: see only gradle-build job in CI/CD, since grade-test will be skipped (as current push does not involve a pull/merge request)

Some info on Docker

- Assembly of linux functions/security applications
- Better performance than VM because doesn't copy entire OS, but shares linux kernel with host
- Good for run automated tasks in standardized environment
- Dependency hell – isolate services
- Docker image – snapshot of Docker at certain point in time, Docker container – instance of docker image (think of a digital photograph as Docker image, printout of photo as Docker container)

Activity 3

Aim: use the nginx framework as a web server within a Docker container and display your custom index.html page

1. Run the nginx Docker image:	`docker run --name nginx-server -d -p 8080:80 nginx` This command does the following: --name nginx-server: Names the container nginx-server. -d: Runs the container in detached mode. -p 8080:80: Maps port 8080 on the host to port 80 on the container. nginx: Specifies the nginx Docker image.
2. Access the web server:	Open a web browser and navigate to http://localhost:8080. You should see the default nginx welcome page.
3. Create an index.html page on your local computer:	``` <!DOCTYPE html> <html lang="en"> <head> <meta charset="UTF-8"> <meta name="viewport" content="width=device-width, initial-scale=1.0"> <title>My Custom Page</title> </head> <body> Welcome to My Custom Page! </body> </html> ```
4. Mount the local index.html into the nginx container:	First, stop and remove the previous nginx container: ``` docker stop nginx-server docker rm nginx-server ``` Now, run the nginx Docker image again, but this time mount the local index.html: ``` docker run --name nginx-server -d -p 8080:80 -v /path/to/your/index.html:/usr/share/nginx/html/index.html nginx ``` Replace /path/to/your/index.html with the actual path to your index.html file.
5. Access the web server again:	Navigate to http://localhost:8080 in your web browser. This time, you should see the content of your custom index.html page.

Note: make sure to include the path to default nginx Docker image as well

Final command looks something like

```
docker run --name nginx-server -d -p 8080:80 -v ./index.html:/usr/share/nginx/html/index.html nginx
```

Activity 4

Aim: run nginx container using Docker compose and mount index.html file

Note: background information may be required

1. Create a Docker Compose File:	Create a file named docker-compose.yml in the directory where your index.html is located.

2. Add the Following Content to docker-compose.yml:	```
	version: '3'
	
	services:
	  nginx-server:
	  image: nginx
	  ports:
	  - "8080:80"
	  volumes:
	  - ./index.html:/usr/share/nginx/html/index.html
	``
	
	Explanation:
	• version: '3': Specifies the version of the Docker Compose file format.
	• services: Defines the services to be run.
	• nginx-server: The name of the service.
	• image: nginx: Specifies the nginx Docker image.
	• ports: Maps port 8080 on the host to port 80 on the container.
	• volumes: Mounts the local index.html file to the location inside the container where nginx serves the default page.

3. Start the nginx Container using Docker Compose:	In the directory where your docker-compose.yml and index.html files are located, run: docker-compose up -d
4. Access the Web Server:	Open a web browser and navigate to http://localhost:8080. You should see the content of your custom index.html page.
5. Stop the nginx Container:	docker-compose down

Note: may need to remove nginx-sever again (using docker stop and docker rm) before able to do step 3

Extension task

Most students probably won't get up to this. But here's a general outline of how to approach:

1. Prerequisites:	Ensure you have Docker installed on your machine (or VM if you're using a cloud provider). Have a GitLab account and access to a GitLab project.
2. Install GitLab Runner:	Depending on your OS, the installation process may vary. Here's a general approach: For Debian/Ubuntu: ``` curl -LJO "https://gitlab-runner-downloads.s3.amazonaws.com/latest/deb/gitlab-runner_amd64.deb" sudo dpkg -i gitlab-runner_amd64.deb ``` For Red Hat/CentOS: ``` curl -LJO "https://gitlab-runner-downloads.s3.amazonaws.com/latest/rpm/gitlab-runner_amd64.rpm" sudo rpm -i gitlab-runner_amd64.rpm ``` For other OS or manual installation, refer to the official documentation.
3. Register the GitLab Runner:	Navigate to your GitLab project. Go to Settings > CI/CD.

	Under Runners, find the Set up a specific Runner manually section. Note down the URL and the registration token. On your machine, run: `sudo gitlab-runner register` Follow the prompts: • Enter the coordinator URL (from GitLab). • Enter the registration token (from GitLab). • Enter a description for the runner. • Enter tags (optional but useful for specific jobs). • Choose the executor (e.g., docker). • If you chose docker, specify the default Docker image (e.g., alpine:latest).
4. Start the GitLab Runner:	`sudo gitlab-runner start`
5. Verify Runner Status:	Back in your GitLab project, under Settings > CI/CD > Runners, you should now see your runner listed as active.
6. Configure .gitlab-ci.yml:	In your project, ensure that the .gitlab-ci.yml file uses the tags you specified during registration (if any) to ensure jobs run on your self-hosted runner.
7. Run CI/CD Pipelines:	Whenever you push changes to your GitLab repository or create merge requests, the CI/CD pipeline will trigger, and your self-hosted runner will pick up and execute the jobs.
8. Maintenance:	Regularly check for updates to the GitLab Runner software and update as needed. Monitor the resources on your machine or VM, especially if running intensive CI/CD tasks.