2025

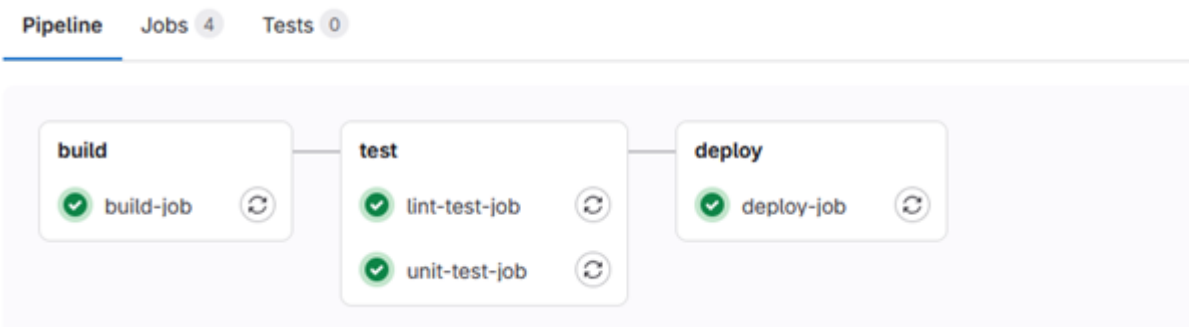
ბიზნესისა და ტექნოლოგიების უნივერსიტეტი

ვერსიონირება და უწყვეტი ინტეგრაცია

თემა მეექვსე და მეშვიდე:
Gitlab Pipelines

Pipelines

GitLab ფაიფლაინები წარმოადგენს ავტომატიზაციის მექანიზმს, რომელიც GitLab-ის CI/CD სისტემის მეშვეობით უზრუნველყოფს კოდის ავტომატურ აგებას, ტესტირებას, განთავსებას და სხვა პროცესებს. ეს არის YAML-ფორმატში აღწერილი სცენარები, რომლებიც განსაზღვრავენ, როგორ უნდა შესრულდეს სხვადასხვა დავალება ციკლის განმავლობაში. GitLab ფაიფლაინი იწერება სპეციალურ კონფიგურაციურ ფაილში .gitlab-ci.yml, რომელიც ინახება პროექტის მთავარ დირექტორიაში. ამ ფაილში აღწერილია, თუ რა უნდა გაკეთდეს სხვადასხვა ეტაპზე, რა გარემოში, და რა დამოკიდებულებები აქვს ერთ დავალებას მეორეზე.

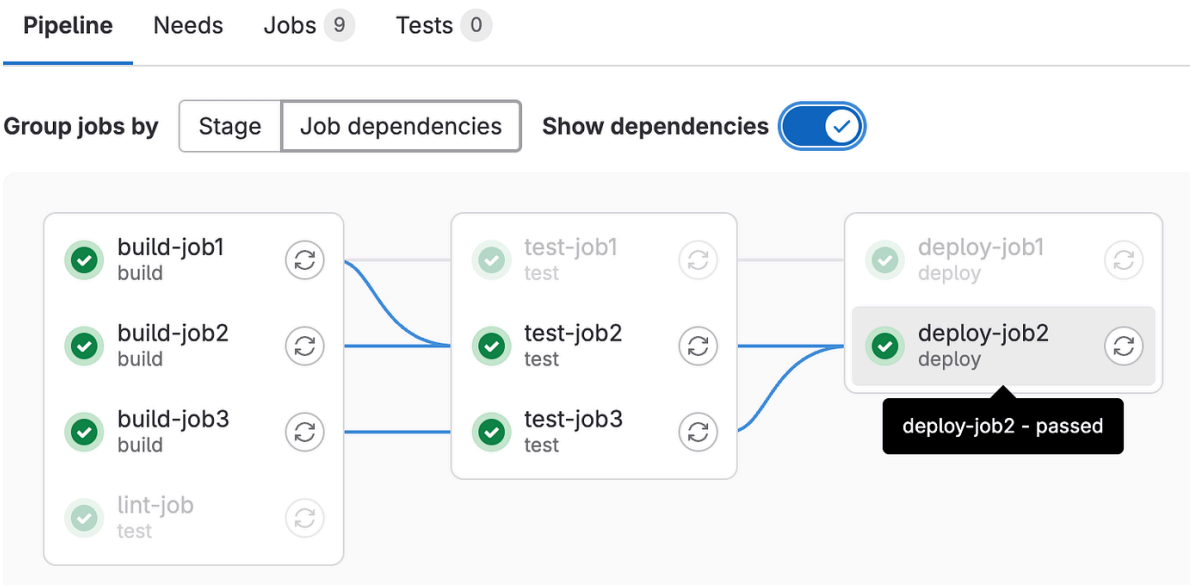


ასე გამოიყურება აწყობილი ფაიფლაინი, რომელიც შედგება სამი ეტაპისგან (stage-ისგან), ესენია: build, test და deploy.
ფაიფლაინში გვაქვს ოთხი job-ი:

- Build-job - რომელიც build stage-შია დამატებული
- Lint-test-job - რომელიც test stage-შია დამატებული
- Unit-test-job - რომელიც test stage-შია დამატებული
- Deloy-job - რომელიც deploy stage-შია დამატებული

აღნიშნულ ფაიფლაინში დამოკიდებულება stage-ებს შორისაა, რაც ნიშნავს იმას რომ ყოველი stage-ი დაიწყება მაშინ და მხოლოდ მაშინ როცა წინა stage-ები წარმატებით დაასრულებენ მუშაობას. აღნიშნულ მაგალითზე შეგვიძლია ვთქვათ რომ test stage-ი დაიწყებს მუშაობას მხოლოდ მაშინ როცა build stage-ი წარმატებით დაასრულებს მუშაობას, ხოლო deploy stage-ი დაიწყებს მუშაობას მაშინ როდესაც test stage-ის ორივე job-ი წარმატებით დაასრულებენ თავის მუშაობას.

დამოკიდებულება შეიძლება იყოს არა მარტო stage-ებს არამედ job-ებს შორისაც.



აღნიშნულ მაგალითში ლურჯი ზოლებით გამოკვეთილია დამოკიდებულებები (dependencies) job-ებს შორის. იმისათვის რომ გაეშვას test-job2 საჭიროა რომ შესრულდეს build-job1 და build-job2, ხოლო, test-job3-ის ასამუშავებლად საჭიროა build-jobs3-ის მუშაობის წარმატებით დასრულება. Deploy-job2-ის ასამუშავებლად კი საჭიროა test-job2-ის და test-job3-ის მუშაობის წარმატებით დასრულება.

Pipeline-ის სტრუქტურა მოიცავს:

- **Stages** - Stage-ები განსაზღვრავს ფაიფლაინის სტრუქტურულ საფეხურებს. Stage-ების კლასიკური მაგალითია (Build, Test, Deploy)
- **Jobs** - Job-ი არის კონკრეტული ეტაპი, რომელიც ასრულებს განსაზღვრულ მოქმედებებს (ბრძანებებს). იმისათვის რომ მივუთითოთ თუ რომელ stage-ზე უნდა ამოქმედდეს ესა თუ ის job-ი, job-ში უნდა მივუთითოთ პარამეტრი “stage: stage-ის სახელი:
- **Runners** - GitLab Runner არის აგენტი, რომელიც ასრულებს ფაიფლაინის დავალებებს. ის შეიძლება იყოს GitLab-ის მიერ მოწოდებული Shared Runner-ი, ან self-hosted
- **Script** - განსაზღვრავს კონკრეტულ ბრძანებებს, რომლებიც უნდა შესრულდეს job-ის ფარგლებში
- **Artifacts** - ეს არის ფაილები, რომლებიც იქმნება ერთ ეტაპზე და საჭიროა შემდეგ ეტაპზე გადასაცემად
- **Needs** - პირობები რომლებიც უნდა შესრულდეს იმისათვის რომ მიმდინარე job-ი ამუშავდეს. მაგალითად job2-ის ამუშავების პირობა

შეიძლება იყოს იმავე stage-ის job1-ის წარმატებით მუშაობის დასრულება

- **Rules** - ბრძანებები რომელთა მეშვეობითაც შესაძლებელია პირობების გაწერა, მაგალითად თუ როდის უნდა ამუშავდეს კონკრეტული job-ი ან პირიქით არ ამუშავდეს. მაგალითად თუ საჭიროა job-ის ამუშავება მხოლოდ main ბრენჩზე დაფუშვისას

- **Cache** - cache-ის საშუალებით შესაძლებელია ფაიფლაინის მუშაობის დასრულების შემდეგ შედეგის ან მოდულების შენახვა და ასევე მათი გადაცემა მეორე ფაიფლაინზე

განვიხილოთ მარტივი ფაიფლაინი,

```
image: node:18  # Ubuntu-ზე დაფუძნებული Node.js Docker იმიჯი

stages:
    - build
    - test
    - deploy

cache:
    key: npm-cache
    paths:
        - node_modules/

before_script:
    - apt-get update -y

build_app:
    stage: build
    script:
        - echo "Installing dependencies on Ubuntu..."
        - npm install
        - echo "Building application..."
        - npm run build

test_app:
    stage: test
    script:
        - echo "Running tests on Ubuntu..."
        - npm test
    needs: [build_app]



deploy_prod:
    stage: deploy
```

```
script:
        - echo "Deploying to production on Ubuntu..."
        - chmod +x ./deploy.sh
        - ./deploy.sh
rules:
        - if: '$CI_COMMIT_BRANCH == "main"'
```

მაგალითში ვხედავთ რომ გვაქვს მითითებული docker image რომელიც შეიცავს node.js -ს

გვაქვს სამი stage - build, test და deploy

cache რომელშიც ვინახავთ node-module-ს დროის დასაზოგად

before_script-ი რომელიც ანახლებს პაკეტებს მანამ სანამ ძირითად სკრიპტზე გადავა

build_app არის job-ი რომელიც build stage-ის ნაწილია და script-ში გაწერილი აქვს ბრძანებები რომლებიც უნდა შეასრულოს

test_app არის job-ი რომელიც test stage-ის ნაწილია და script-ში გაწერილი აქვს ბრძანებები რომლებიც უნდა შეასრულოს, ასევე მას აქვს მითითებული რომ უნდა გაეშვას მხოლოდ build_app job-ის წარმატებით დასრულების შემდეგ

deploy_prod არის job-ი რომელიც deploy stage-ის ნაწილია და script-ში გაწერილი აქვს ბრძანებები რომლებიც უნდა შეასრულოს, ასევე მითიტებულია რამოდენიმე პირობა: if: '$CI_COMMIT_BRANCH == "main"' - ამოწმებს მოხდა თუ არა ქომითი main ბრენჩზე და გაეშვება მხოლოდ იმ შემთხვევაში თუ main ბრენჩზე იყო გაშვებული ფაიფაინი