THOMAS MORE

Lab 3.2

Adding Quality Gates to the Pipeline

Toepasbare Cloud Security voor IT-Professionals

2025 - 2026

Contents

1. Adding Quality Gates to the Pipeline

1.1. Context

In Lab 3.1, you deployed the basic backend app with a minimal pipeline. Now you'll add quality gates before deployment: automated checks that prevent broken or untested code from being deployed.

Quality gates include:
- Type checking: Verify TypeScript code compiles without type errors
- Unit tests: Test individual functions in isolation
- Integration tests: Test HTTP endpoints with a running server

These checks run before building the artifact, ensuring only tested code proceeds to deployment.

The tests already exist in the repository:
- `apps/backend/src/config.test.ts` — Unit tests for configuration validation
- `apps/backend/src/app.test.ts` — Unit tests for app creation and route registration
- `apps/backend/src/app.integration.test.ts` — Integration tests for HTTP endpoints

Your task: Run these tests locally, understand what they validate, then integrate them into the pipeline.

1.2. Learning goals

By the end of this lab, you should be able to explain:

- Why type checking runs before tests, and why both run before security scans.
- The difference between unit tests and integration tests, and when to use each.
- How the test pyramid guides test design (many fast unit tests, fewer slow integration tests).
- Why quality gates should fail the pipeline (unlike security scans which start in audit mode).

1.3. Estimated time

60 minutes

Suggested timing:

Time	Activity
10 minutes	Run tests locally and understand what they validate
15 minutes	Review test code (unit vs integration)
15 minutes	Run the pipeline with quality gates enabled
10 minutes	Intentionally break a test and observe pipeline failure
10 minutes	Reflection questions

1.4. Before you start

Make sure you have:

- [ ] Repository cloned locally with `course-cloud-security-app/apps/backend/` directory

- [ ] Node.js 22.x installed (`node --version`)
- [ ] Dependencies installed (`npm install` in backend directory)
- [ ] Azure DevOps pipeline configured (from Lab 3.1)

2. Run Tests Locally

Navigate to the backend application directory:

```
cd <BACKEND_APP_FOLDER>
```

> **Note**
>
> Cleanup Note: The test output files `unit-test-results.tap` and `integration-test-results.tap` are generated during pipeline execution. You should add these files to `.gitignore` if you don't want them committed, or commit them if your team prefers to track test output formats.

2.1. Type Checking

If needed install packages first with `npm install` Run the TypeScript type checker:

```
npm run typecheck
```

What is the output? (Does it pass or show errors?)

Your answer:

> **Question**
>
> What does `tsc --noEmit` do? Why not just run `npm run build`?

Your answer:

2.2. Unit Tests

Run only the unit tests:

```
npm run test:unit
```

How many tests pass?

> **Note**
>
> If they all fail, you might be missing something. Check the error and see if you can figure it out.

Your answer:

List the test files that are executed:

Your answer:

2.3. Integration Tests

Run only the integration tests:

```
npm run test:integration
```

How many tests pass?

Your answer:

What is the difference in execution time between unit and integration tests?

Your answer:

2.4. Run All Tests

Run all tests together:

```
npm test
```

Total number of passing tests:

Your answer:

3. Review Test Code

3.1. Unit Test: Configuration Validation

Open `apps/backend/src/config.unit.test.ts`.

What does the test `"validates required environment variables"` check?

Your answer:

What happens if `FRONTEND_ORIGIN` environment variable is missing?

Your answer:

3.2. Integration Test: HTTP Endpoints

Open `apps/backend/src/app.integration.test.ts`.

What does the test `"GET /api/health returns 200"` do?

Your answer:

How does this test differ from the unit test in `app.test.ts`?

Your answer:

4. The Test Pyramid

In your backend application:

- Unit tests: Test `config.ts` validation, route registration (no HTTP server)
- Integration tests: Test HTTP endpoints with running server (no database yet)
- E2E tests (not implemented): Would test full frontend → backend → database flow

> **Question**
>
> Why have many unit tests but fewer integration tests? Why not just write integration tests for everything?

Your answer:

5. Integrate Tests into the Pipeline

The starter pipeline already provisions infrastructure, builds the app, and deploys it. Now add a `Validate` stage between infrastructure provisioning and `Build` so the app is only built and deployed when type checking and tests pass.

5.1. Add the Validate Stage

Open `azure-pipelines.yml` and add this stage above the existing `Build` stage:

```yaml
- stage: Validate
  displayName: 'Quality Gates'
  dependsOn: ProvisionDevInfra
  condition: succeeded()
  jobs:
    - job: QualityGates
      displayName: 'Type Check and Tests'
      pool:
        vmImage: 'ubuntu-latest'
      steps:
        - task: UseNode@1
          displayName: 'Install Node.js'
          inputs:
            version: '$(nodeVersion)'

        - script: |
            echo "appRootPath = $(appRootPath)"
            pwd
            ls -la
            node -v
            npm -v
            npm ci
          displayName: 'Install dependencies'
          workingDirectory: '$(appRootPath)'

        - script: |
            npm run typecheck
          displayName: 'TypeScript type check'
          workingDirectory: '$(appRootPath)'

        - script: |
            set -euxo pipefail
            mkdir -p test-results
            npm run test:unit:junit
          displayName: 'Run Unit tests'
          workingDirectory: '$(appRootPath)'
          env:
            NODE_ENV: test
            FRONTEND_ORIGIN: http://localhost:5173

        - script: |
            set -euxo pipefail
            npm run test:integration:junit
          displayName: 'Run Integration tests'
          workingDirectory: '$(appRootPath)'
```

```
    env:
      NODE_ENV: test
      FRONTEND_ORIGIN: http://localhost:5173

  - task: PublishTestResults@2
    displayName: 'Publish test results'
    condition: always()
    inputs:
      testResultsFormat: 'JUnit'
      testResultsFiles: '$(appRootPath)/test-results/*.xml'
      testRunTitle: 'Unit & Integration Tests'
      mergeTestResults: true
      failTaskOnFailedTests: true
```

List the 3 main script steps in the `QualityGates` job:

Your answer:

"Why does type checking run BEFORE tests? What's the benefit of failing fast on type errors?

Your answer:

What does `PublishTestResults@2` do? Why publish test results even if they pass?

Your answer:

5.2. Make Build Depend on Validate

Find the existing `Build` stage and update it so it only runs after `Validate` succeeds:

```yaml
  - stage: Build
    displayName: 'Build Basic App'
    dependsOn: Validate
    condition: succeeded()
```

This changes the pipeline flow from:

```
ProvisionDevInfra → Build → DeployDev
```

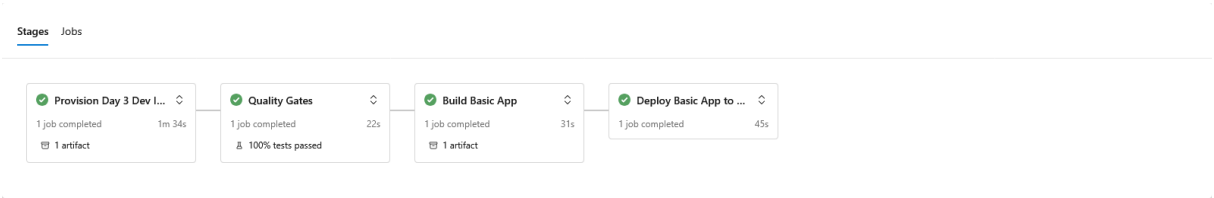
to:

```
ProvisionDevInfra → Validate → Build → DeployDev
```

6. Run the Pipeline with Quality Gates

Commit the updated starter pipeline and push to trigger the pipeline:

```
git add azure-pipelines.yml
git commit -m "feat: add quality gates pipeline (Lab 3.2)"
git push
```



Navigate to Azure DevOps → Pipelines → Your pipeline run.

6.1. Observe the Validate Stage

Does the `QualityGates` job complete successfully?

Your answer (Yes/No):

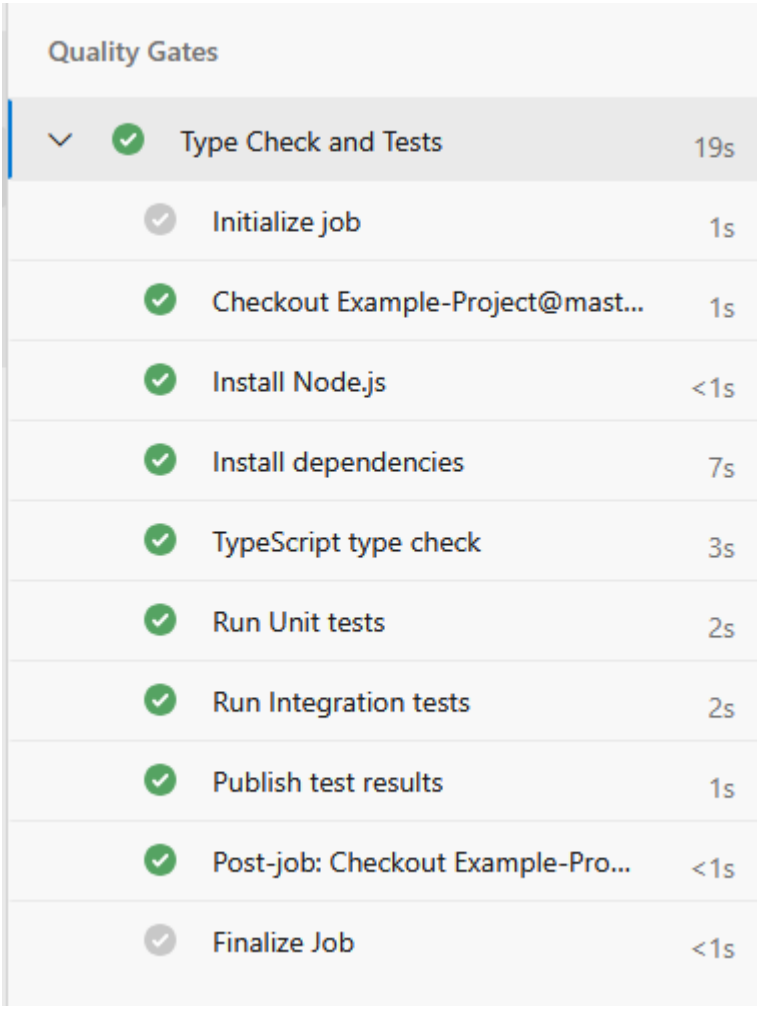
What is the execution time for the `QualityGates` job?

Your answer:

> **Note**
>
> In Lab 3.3, you'll add security scanning jobs that run in parallel with QualityGates. For now, QualityGates is the only job in the Validate stage.

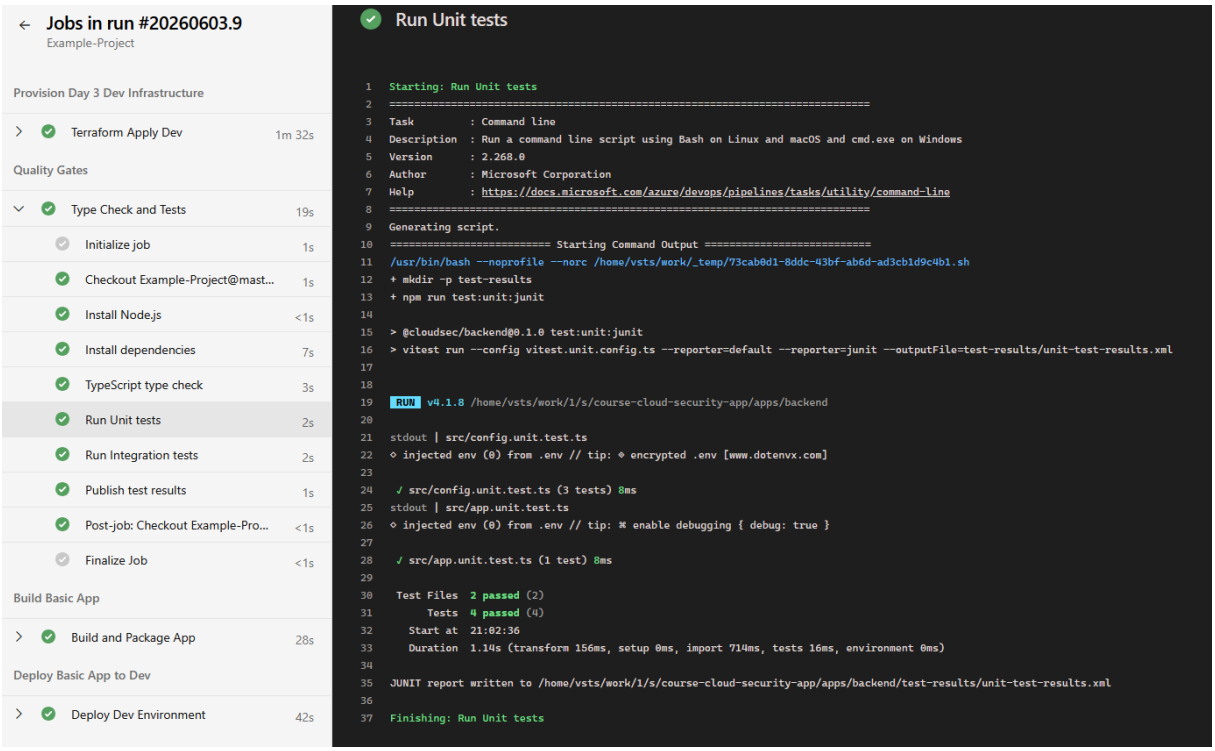
6.2. Review the Logs

Click on the `QualityGates` job and expand the steps.



What is the output of the `typecheck` step?

Your answer:



What is the output of the `test:unit` step?

Your answer:

7. Intentionally Break a Test

7.1. Introduce a Type Error

Edit `apps/backend/src/config.ts` and change this line:

```
// Before
const port = parseInt(process.env.PORT || '3000', 10);

// After (introduce type error)
const port: string = parseInt(process.env.PORT || '3000', 10);
```

Run type checking locally:

```
npm run typecheck
```

What error does TypeScript report?

Your answer:

7.2. Commit and Push

```
git add apps/backend/src/config.ts
git commit -m "test: introduce type error"
git push
```

7.3. Observe Pipeline Failure

Navigate to Azure DevOps and watch the pipeline run.

Does the `QualityGates` job fail?

Your answer (Yes/No):

Do the security scanning jobs still run, or does the pipeline stop immediately?

Your answer:

Does the Build stage run?

Your answer (Yes/No and why):

> **Question**
> Why should quality gates FAIL the pipeline instead of using `continueOnError: "true"` like security scans?

Your answer:

7.4. Fix the Error

Revert the type error:

```
git revert HEAD
git push
```

Verify the pipeline passes again.

8. Reflection Questions

8.1. Question 1: Why Quality Before Security?

The pipeline runs quality gates (typecheck, tests) before security scans (Semgrep, Trivy).

Why is this order important?

Your answer:

8.2. Question 2: Unit vs Integration Test Trade-offs

You have a function that queries the database and returns user data.

Would you test this with a unit test or integration test? What are the trade-offs?

Your answer:

8.3. Question 3: Should Tests Run in Deployment Stages?

Currently, tests run in the Validate stage (before building the artifact).

Should tests also run in the DeployDev stage (after deploying to dev environment)?

Your answer:

8.4. Question 4: Test Coverage

How do you know if your tests cover enough of the codebase?

(Hint: Look up “code coverage” and consider whether 100% coverage guarantees no bugs)

Your answer:

9. Summary

In this lab, you:

- Ran type checking, unit tests, and integration tests locally
- Understood the test pyramid (many fast unit tests, fewer slow integration tests)
- Integrated tests into the pipeline as quality gates
- Observed pipeline failure when tests fail (fail-fast principle)
- Learned why quality gates run before security scans (speed and clarity)

> **Note**
>
> In the next lab (Lab 3.3), you will add security scanning (Semgrep and Trivy) to detect code vulnerabilities and dependency CVEs.

10. Next Steps

- [ ] All tests pass locally (`npm test`)
- [ ] Pipeline quality gates pass in Validate stage
- [ ] Understand when to write unit vs integration tests
- [ ] Know why quality gates should fail the pipeline

Continue to **Lab 3.3: Security Scanning with Semgrep and Trivy**.