LAB 1: BUILDING A CI/CD PIPELINE WITH GITHUB ACTIONS

Modern JavaScript ES Modules • Node.js 24 • Yarn 4 • Express 5 • Mocha • Chai • Supertest • ESLint • c8

Student Lab Handout | Suggested duration: 3 hours

Team: _____ Class: _____ Date: _____

Student name: _____

_____

1. Lab Overview

In this lab, you will create a small Node.js web application from an empty project folder and build a Continuous Integration (CI) pipeline using GitHub Actions. You will create every required file yourself, install dependencies with Yarn 4, write automated tests, measure code coverage locally, and configure CI to run linting, tests, and coverage checks.

The application uses modern JavaScript ES Modules, Express 5, Mocha, Chai, Supertest, ESLint, and c8 for coverage reporting. The root endpoint (GET /) should return the text “Hello, GitHub Actions!”.

2. Learning Outcomes

- Explain Continuous Integration (CI), Continuous Delivery (CD), and Continuous Deployment.
- Create a Node.js project using ES Modules and Yarn 4.
- Write and test an HTTP endpoint with Express, Mocha, Chai, and Supertest.
- Run ESLint and fix code-quality issues.
- Measure statement, branch, function, and line coverage locally.
- Configure GitHub Actions to run linting, tests, and code coverage on pushes and pull requests.
- Submit a repository, evidence, and a short technical reflection.

3. Required Tools

- Node.js 24 or later.
- Git and a GitHub account.
- A code editor, such as Visual Studio Code.
- Internet access to install packages and push to GitHub.

Check your environment:

```bash
node --version
git --version
```

4. What You Will Build

You will create the following project structure. Create the folders and files exactly as shown; do not download or use a starter ZIP.

```text
ci-cd-lab/
├── .github/
│   └── workflows/
│       └── ci.yml
├── test/
│   └── test.js
├── .gitignore
├── .yarnrc.yml
├── app.js
```

```
├── eslint.config.js
├── package.json
├── README.md
└── server.js
```

Yarn will generate yarn.lock after you install dependencies. Commit that file to Git along with your source files.

5. Implementation

Step 1 — Create the Project Folder

Create a new directory and enter it. You can use a terminal or create the folder in your editor.

```bash
mkdir ci-cd-lab
cd ci-cd-lab
git init
```

Create the test and GitHub Actions folders:

```bash
mkdir test
mkdir -p .github/workflows
```

If your terminal does not support mkdir -p, create the folders manually in your editor.

Step 2 — Create package.json

Create a file named package.json in the project root and paste the following content. This file defines the project type, package manager, dependencies, and commands used in the lab.

```json
{
  "name": "ci-cd-lab",
  "version": "1.0.0",
  "private": true,
  "type": "module",
  "packageManager": "yarn@4.18.1",
  "engines": {
    "node": ">=24"
  },
  "scripts": {
    "start": "node server.js",
    "test": "mocha",
    "lint": "eslint .",
    "coverage": "c8 mocha"
  },
  "dependencies": {
    "express": "^5.1.0"
  },
  "devDependencies": {
    "@eslint/js": "^10.0.1",
    "c8": "^10.1.3",
    "chai": "^6.3.0",
    "eslint": "^10.13.0",
```

```
    "globals": "^16.5.0",
    "mocha": "^12.0.3",
    "supertest": "^7.3.1"
  }
}
```

Step 3 — Create app.js

Create app.js in the project root. This file defines the Express application but does not start a listening server. Keeping these responsibilities separate makes the application easier to test.

```javascript
import express from 'express';

const app = express();

app.get('/', (req, res) => {
  res.status(200).send('Hello, GitHub Actions!');
});

export default app;
```

The import statement uses ES Modules. The GET / route returns HTTP status 200 and a greeting. The export default statement lets the test file import the app.

Step 4 — Create server.js

Create server.js in the project root:

```javascript
import app from './app.js';

const port = process.env.PORT ?? 3000;

app.listen(port, () => {
  console.log(`Server listening on port ${port}`);
});
```

This file starts the server. It uses the PORT environment variable when provided; otherwise it listens on port 3000.

Step 5 — Create the Test File

Create test/test.js. The two tests check the HTTP status and response text returned by GET /.

```javascript
import { expect } from 'chai';
import request from 'supertest';
import app from '../app.js';

describe('GET /', () => {
  it('returns HTTP 200', async () => {
    const response = await request(app).get('/');
    expect(response.status).to.equal(200);
  });
```

```
  it('returns the expected greeting', async () => {
    const response = await request(app).get('/');
    expect(response.text).to.equal('Hello, GitHub Actions!');
  });
});
```

Mocha provides describe() and it(). Chai provides expect() assertions. Supertest sends HTTP requests directly to the Express app, so the test does not need you to start server.js manually.

Step 6 — Create eslint.config.js

Create eslint.config.js in the project root. This uses ESLint's flat configuration format and enables Node.js and Mocha globals.

```javascript
import js from '@eslint/js';
import globals from 'globals';

export default [
  {
    ignores: ['coverage/**', 'node_modules/**', '.yarn/**'],
  },
  js.configs.recommended,
  {
    files: ['**/*.js'],
    languageOptions: {
      ecmaVersion: 'latest',
      sourceType: 'module',
      globals: {
        ...globals.node,
        ...globals.mocha,
      },
    },
  },
];
```

ESLint checks for common coding mistakes and consistency issues. If your environment reports that your installed ESLint version does not support the configured runtime, check the Node.js version and installed package versions.

Step 7 — Install Dependencies

Run the following commands from the ci-cd-lab project directory:

```bash
yarn --version
yarn install
```

Confirm that Yarn reports version 4 and that yarn.lock is generated. Keep the packageManager field and CI activation command aligned.

If installation fails, check your Node.js version and network access. Do not continue to the GitHub Actions step until dependencies install successfully.

Step 8 — Run the Application Locally

Start the app:

```bash
yarn start
```

Open http://localhost:3000/ in a browser. The page should display:

```text
Hello, GitHub Actions!
```

Stop the server with Ctrl+C after checking the endpoint.

Step 9 — Run ESLint and Tests Locally

Run each command from the project root:

```bash
yarn lint
yarn test
```

Both commands should finish successfully. The test output should report two passing tests. If a command fails, read the error output, correct the relevant file, and run both commands again.

Experiment: temporarily change the greeting in app.js, then run yarn test. Observe why the response-text test fails. Restore the original greeting and rerun the tests. Do not commit the deliberately broken version.

Step 10 — Measure Code Coverage Locally

Code coverage estimates which parts of your source code were exercised by your tests. This lab uses c8, which collects coverage for Node.js using the built-in V8 coverage engine.

Run the coverage command:

```bash
yarn coverage
```

The command runs Mocha under c8 and prints a coverage table with metrics such as statement, branch, function, and line coverage. It also creates a coverage/ directory containing an HTML report.

Open coverage/index.html in your browser to inspect the report. You can also run the following command if your operating system supports it:

```bash
open coverage/index.html      # macOS
start coverage/index.html     # Windows Command Prompt
xdg-open coverage/index.html  # many Linux desktops
```

Use the command appropriate for your operating system. The HTML report is generated locally; the coverage/ folder is ignored by Git so generated reports are not committed.

Interpretation: 100% coverage means the measured code was exercised by tests, not that the application is bug-free or that every possible behavior has been tested. The current tests exercise the root endpoint. Add more routes and tests to explore how coverage changes.

Optional extension: configure a coverage threshold so the command fails if coverage falls below a chosen level. If you do this, explain the threshold and ensure your test suite can meet it.

Step 11 — Create the GitHub Actions Workflow

Create .github/workflows/ci.yml. GitHub Actions runs this workflow on pushes to main and pull requests targeting main. It installs dependencies, runs linting and tests, then generates the coverage report.

```yaml
name: CI

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

jobs:
  quality:
    runs-on: ubuntu-latest
    steps:
      - name: Check out repository
        uses: actions/checkout@v4

      - name: Set up Node.js
        uses: actions/setup-node@v4
        with:
          node-version: 24
          cache: yarn

      - name: Install dependencies
        run: yarn install --immutable

      - name: Run ESLint
        run: yarn lint

      - name: Run tests
        run: yarn test

      - name: Generate code coverage
        run: yarn coverage

      - name: Upload coverage HTML report
        if: always()
        uses: actions/upload-artifact@v4
        with:
          name: code-coverage-report
          path: coverage/
          if-no-files-found: ignore
```

The --immutable option tells Yarn not to modify the lockfile during CI. If the workflow fails because the lockfile is missing or out of date, run yarn install locally and commit the updated yarn.lock.

The workflow uploads the HTML coverage report as an artifact. After a run, open the workflow run in GitHub and look for the artifact named code-coverage-report. Download it and open index.html locally to inspect the report.

Step 12 — Initialize Git and Push to GitHub

Create an empty repository on GitHub. If you have already initialized the local project with git init, do not initialize a second repository inside it.

```bash
git status
git add .
```

```
git commit -m "Build application and CI checks"
git branch -M main
git remote add origin https://github.com/YOUR-USERNAME/YOUR-REPOSITORY.git
git push -u origin main
```

Replace the sample URL with your repository URL. If origin already exists, inspect it with git remote -v instead of adding it again. Make sure yarn.lock is committed. Never commit credentials or secrets.

Step 13 — Verify CI and Coverage on GitHub

- Open your GitHub repository and select Actions.
- Open the CI run triggered by your push to main.
- Confirm dependency installation, ESLint, tests, and code coverage steps complete successfully.
- Download the code-coverage-report artifact and open coverage/index.html to inspect the report.
- If any step fails, open the failed step's logs, fix the problem locally, run yarn lint, yarn test, and yarn coverage, then commit and push the fix.

Take screenshots of the successful workflow and the local coverage summary. Your instructor should be able to open your repository and see the workflow run.

Step 14 — Demonstrate CI on a Pull Request

Create a feature branch and make a small improvement. For example, add a GET /health endpoint and a corresponding test. Run all three checks locally before pushing.

```bash
git checkout -b feature/health-route
# Edit app.js and test/test.js
yarn lint
yarn test
yarn coverage
git add app.js test/test.js
git commit -m "Add health endpoint and test"
git push -u origin feature/health-route
```

Open a pull request from feature/health-route into main. Confirm that the CI workflow runs for the pull request and that its checks pass. If you implement the health route, make sure it returns the response you intend and that the test verifies it.