Software Engineering & Development

TP 4 Tutorial — DevOps Lab

Jira, GitHub, VS Code, CI/CD and Node.js

This lab simulates a real DevOps workflow for a small development team. It guides you through the full software development lifecycle : from project planning in Jira, to collaborative coding in GitHub + VS Code, automated builds with GitHub Actions, and quality assurance through unit + integration tests with Vitest and Supertest.

You will work as a team of four developers, each responsible for a different feature of a shared Node.js project.

The source code base is provided in the ZIP archive:
📦 Full_DevOps_Lab.zip

Learning Outcomes

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

- PLAN: Structure a Scrum project in Jira Cloud, create Epics & User Stories,
  and link them to real commits and pull requests through the GitHub for Jira integration.
- CODE: Develop collaboratively in VS Code using Git branches, clear commit messages,
  and pull requests associated with Jira issues.
- BUILD: Configure and analyze a GitHub Actions CI/CD pipeline that automates linting, testing, and coverage verification for every commit and PR.
- TEST: Execute and interpret Vitest + Supertest suites, maintain minimum coverage thresholds (≥ 80 %),
  and ensure each module behaves correctly before merging to the main branch.

Project Overview

Each student in the 4-person team owns one Jira User Story and its corresponding code module in the shared repository.

Student	Jira Story	File in Project	Functionality
A	DEVOPS-1	src/routes/auto/version.route.js	Return application version
B	DEVOPS-2	src/routes/auto/info.route.js	Return runtime info
C	DEVOPS-3	src/routes/auto/boom.route.js	Simulate an API error
D	DEVOPS-4	src/utils/status.js	Return uptime status label

All four stories together form the core API of the application, which will be tested and validated through CI.

Lab Workflow Summary

Step	Phase	Key Tools	Deliverables
1 – PLAN	Project organization	Jira Software Cloud	Epics, User Stories, Jira–GitHub link
2 – CODE	Collaborative development	GitHub, VS Code	Feature branches, commits, PRs
3 – BUILD	Continuous Integration	GitHub Actions	Automated lint + test pipeline
4 – TEST	Quality Assurance	Vitest, Supertest	80 % coverage, green CI checks

Step 1 — PLAN (Jira Cloud)

Structure the DevOps project in **Jira** to manage collaboration, track progress, and link each feature directly to commits, branches, and pull requests in GitHub.
Each student will handle one User Story connected to a real file in the shared Node.js project (see ZIP).

Detailed Instructions

1. Create a Jira Software (Scrum) project

- Go to: https://www.atlassian.com/software/jira
- Choose: **Software → Scrum → Create Project**
- Project name: **DevOps Lab Node.js Project**
- Project key: **DEVOPS** (it will prefix issue IDs like DEVOPS-3)

2. Configure your team board

- Go to **Board Settings → Columns**
- Create the workflow:
- To Do → In Progress → Review → Done
- Assign all 4 students to the same project board.

Each Jira issue (User Story) will move through these columns as commits and PRs progress.

3. Create Epics

Epic	**Description**
API Core Development	Endpoints for /version, /info, /boom, /health
Error Handling	Exception management and global middleware
Monitoring	Application uptime, runtime info, and status labels

Epic	Description
CI/CD Automation	Linting, testing, coverage, and GitHub Actions pipeline

Each User Story (below) should be linked to the **API Core Development** Epic.

4. Add User Stories (4 minimum)

Each User Story corresponds to a **student's assigned feature** in the ZIP project.

Story ID	Description	Acceptance Criteria
DEVOPS-1	Implement /version endpoint	- GET /version returns { "version": "x.x.x" } - HTTP 200 - Integration test passes
DEVOPS-2	Implement /info endpoint	- GET /info returns { name, version, node, uptime } - JSON response - Unit test for helpers
DEVOPS-3	Implement /boom endpoint	- GET /boom triggers a 500 error - Global error handler returns { error: true, message }
DEVOPS-4	Implement formatStatus() utility	- Function returns warming-up, healthy, or steady depending on uptime - Unit test covers all branches

Each story is directly mapped to one file in the **ZIP project**:

Student	Jira Story	File to Edit	Role
A	DEVOPS-1	src/routes/auto/version.route.js	Develop /version route
B	DEVOPS-2	src/routes/auto/info.route.js	Develop /info route
C	DEVOPS-3	src/routes/auto/boom.route.js	Add error simulation route
D	DEVOPS-4	src/utils/status.js	Improve uptime label utility

5. Example of a complete story

DEVOPS-1 — Create /version API endpoint

As a developer,
I want to provide a /version API route,
So that the app can expose its current version number.

Acceptance Criteria:

- GET /version returns a JSON with "version" key

- Response code = 200
- Integration test test/version.test.js must pass
- Commit and branch include DEVOPS-1

Story transitions to “Done” when PR merged and CI passes

6. Integrate Jira with GitHub

1. In Jira → **Apps → Find new apps → Search “GitHub for Jira (Atlassian)”**
2. Install and authorize your GitHub organization.
3. Connect your repository (Full_DevOps_Lab_Project_Final_Commented_EN.zip → once uploaded to GitHub).

Once integration is complete:

- A branch named feature/DEVOPS-2-info-endpoint appears automatically in Jira when pushed.
- Commits like: feat(DEVOPS-2): add /info endpoint

will link automatically to the Jira story.

- Pull Requests titled: DEVOPS-2 | Implement /info route

will appear in the **Development** panel of the corresponding Jira issue.

Step 2 — CODE (VS Code + GitHub)

Organize the collaborative coding phase: each student works on their assigned feature in the **shared Node.js project** (Full_DevOps_Lab_Project_Final_Commented_EN.zip).
The goal is to **implement, test, and commit** changes using **Git**, **VS Code**, and **Jira conventions**.

Detailed Instructions

1. Clone the project from GitHub

Once the teacher uploads the ZIP project to your class GitHub organization:

```
git clone https://github.com/<org>/<repo>.git

cd <repo>
npm install --legacy-peer-deps
npm ci --legacy-peer-deps
```

Expected files in the root:

```
package.json

src/

test/
```

```
.github/workflows/ci.yml
README.md
```

If you see this structure, the project is correctly initialized.

2. Set up VS Code environment

Open the folder in **VS Code**, then install:

Extension	**Purpose**
ESLint	Highlights syntax/style errors
GitLens	Shows commit & branch info
GitHub Pull Requests & Issues	Open PRs directly in VS Code
Jira and Bitbucket (Atlassian Labs)	Link VS Code with Jira issues

3. Create your branch for the assigned User Story

Each student must create a branch from main following the naming rule:

```
git checkout -b feature/DEVOPS-<story-id>-<short-desc>
```

Student	**Jira Story**	**Branch Example**
A	DEVOPS-1	feature/DEVOPS-1-version-endpoint
B	DEVOPS-2	feature/DEVOPS-2-info-endpoint
C	DEVOPS-3	feature/DEVOPS-3-boom-endpoint
D	DEVOPS-4	feature/DEVOPS-4-status-utility

This ensures Jira automatically links your branch to your story.

4. Edit the correct file in the project (ZIP mapping)

Each student works in a specific file inside the ZIP project. Open these paths in VS Code:

Student	**File Path (in ZIP project)**	**Task Description**
A	src/routes/auto/version.route.js	Implement /version endpoint → returns package version
B	src/routes/auto/info.route.js	Implement /info endpoint → returns app name, version, uptime
C	src/routes/auto/boom.route.js	Implement /boom endpoint → simulate 500 error for testing

Student	File Path (in ZIP project)	Task Description
D	src/utils/status.js	Implement formatStatus() → convert uptime (seconds) into labels

💡 **Tip:**
You can open the file explorer in VS Code → src/ → follow this path to reach your file.
All functions and comments are already scaffolded; you only need to verify or enhance the logic.

5. Local build and run

Before committing, verify that the server works:

```
npm run dev
```

Expected output:

```
[server] listening on http://localhost:3000
```

Test your endpoint in the browser or with curl:

```
http://localhost:3000/version

http://localhost:3000/info

http://localhost:3000/boom
```

6. Run tests locally

Each endpoint already has integration and unit tests in the ZIP under test/.

Endpoint	Integration Test File	Unit Test File
/version	test/version.test.js	—
/info	test/info.test.js	test/unit/appInfo.test.js
/boom	test/boom.test.js	test/unit/errorHandler.test.js
/health & Status Utility	test/health.test.js	test/unit/status.test.js

Run all tests:

```
npm test
```

Run with coverage:

```
npm test -- --coverage

Expected coverage:

Lines: ≥ 80%
```

```
Branches: ≥ 70%
```

If a test fails, open the file indicated and fix the logic or test.

7. Stage and commit with Jira reference

Once your feature works and tests pass:

```
git add .

git commit -m "feat(DEVOPS-2): add /info endpoint"

git push origin feature/DEVOPS-2-info-endpoint
```

Rules:

- Always include the Jira key in the commit (feat(DEVOPS-#): …)
- One commit per logical change (e.g., code → test → fix)
- Commits without Jira keys won't link to issues in Jira.

8. Open a Pull Request (PR)

In GitHub:

1. Click **Compare & Pull Request**.
2. Title:

   ```
   DEVOPS-2 | Implement /info endpoint
   ```
3. Add a short description:
   - What was done
   - Which tests were updated or created
   - Screenshot (optional)
4. Assign another team member for review.
5. Merge only when CI pipeline passes (GitHub Actions).

9. Common commands for everyone

Action	**Command**	**Description**
Start server	npm run dev	Launch app on http://localhost:3000
Run tests	npm test	Execute unit + integration tests
Run coverage	npm test -- --coverage	Generate coverage report in /coverage/
Lint code	npm run lint	Check code style and syntax
View coverage report	open coverage/index.html	Opens HTML report locally

10. Expected end of Step 2 deliverables

Item	Expected Result
Branch created per student (feature/DEVOPS-X-...)	✅
Correct file modified based on assignment	✅
Code runs locally (npm run dev)	✅
Tests pass and coverage ≥ 80 %	✅
Commit contains Jira key and description	✅
Pull request created and linked to Jira story	✅

Step 3 — BUILD (GitHub Actions CI/CD)

Automate testing, linting, and code validation with **GitHub Actions**.
Each push or Pull Request should automatically trigger the CI pipeline to ensure code quality and coverage before merging.

This step connects **local builds** (npm run lint, npm test) with the **CI pipeline** defined in the ZIP project under: .github/workflows/ci.yml

Detailed Instructions

1. Locate the CI configuration file

In your ZIP project, open the following file:

```
.github/
└── workflows/
    └── ci.yml
```

This file defines the **continuous integration workflow** that runs automatically on GitHub every time code is pushed or a Pull Request is created.

2. Understand the CI workflow logic

The file .github/workflows/ci.yml already contains a complete workflow:

name: CI

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

jobs:
  build-and-test:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Setup Node
        uses: actions/setup-node@v4
        with:
          node-version: 20
          cache: npm

      - name: Install dependencies
        run: npm ci

      - name: Lint
        run: npm run lint

      - name: Test (with coverage)
        run: npm test -- --coverage

      - name: Upload coverage artifact
        uses: actions/upload-artifact@v4
        if: always()
        with:
```

```
        name: coverage-report

        path: coverage
```

💡 **Explanation (line by line):**

Section	**Purpose**
on:	Triggers the workflow when code is pushed or a PR is opened on main
jobs:	Defines a job named build-and-test
runs-on:	Uses Ubuntu (Linux) as the runner environment
checkout@v4	Clones your GitHub repository into the CI environment
setup-node@v4	Installs Node.js (v20) and enables caching for faster builds
npm ci	Installs all dependencies defined in package.json
npm run lint	Verifies code formatting and syntax consistency
npm test -- --coverage	Runs all tests (unit + integration) and generates coverage report
upload-artifact	Saves the coverage folder as a downloadable artifact in GitHub Actions

3. Verify local consistency before pushing

Always make sure the CI will succeed by testing locally first:

```
npm run lint

npm test -- --coverage
```

Expected result:

Tests: passed

Coverage summary: Lines ≥ 80%, Branches ≥ 70%

If you get lint errors, fix them before pushing.

4. Push your branch to trigger the workflow

git push origin feature/DEVOPS-2-info-endpoint

Then, go to your repository on GitHub → **Actions tab**.

You should see the workflow **CI** starting automatically:

build-and-test — in progress

Once complete:

- ✅ **Green check** → all steps passed
- ❌ **Red cross** → one or more tests failed or coverage too low

5. Validate the test coverage thresholds

Open the file package.json inside your project root.
Scroll to the "vitest" section — it enforces minimum coverage levels:

```
"vitest": {
  "coverage": {
    "provider": "v8",
    "all": true,
    "include": ["src/**/*.js"],
    "lines": 80,
    "functions": 80,
    "branches": 70,
    "statements": 80
  }
}
```

If the coverage is below these thresholds, CI will fail automatically.

To fix it:

- Add more unit tests in test/unit/*.test.js
- Improve logic coverage (e.g., test negative paths in status.js)
- Rerun: npm test -- --coverage locally until you reach ≥80%

6. Download and check the coverage artifact (on GitHub)

After the CI completes:

1. Go to **Actions → CI → Summary**
2. Scroll down to **Artifacts**
3. Click **coverage-report.zip** → Download
4. Extract it and open index.html to visualize the coverage per file.

You should see:

- Green coverage bars (≥80%)
- Each src/ file listed with its test score

7. Common troubleshooting

Problem	Possible Cause	Fix
CI fails on "npm run lint"	ESLint found style issues	Run npm run lint --fix locally
CI fails on "npm test"	Some tests failed	Check console output, re-run locally
Coverage < 80%	Missing branch or negative tests	Add more unit tests
"Actions not triggered"	Wrong branch (not main or PR)	Merge PR into main or rebase
"Permission denied"	Repo not authorized in Jira/GitHub app	Check app integration permissions

8. Verify CI results in Jira

Once the workflow finishes successfully, go to Jira → open your User Story (e.g., DEVOPS-2):

Under the **Development** panel, you should see:

- Branch linked (feature/DEVOPS-2-info-endpoint)
- 1 commit (feat(DEVOPS-2): add /info endpoint)
- 1 Pull Request (merged)
- **Build status:** ✓ Passed (from GitHub Actions)

This confirms the end-to-end integration between **Jira**, **GitHub**, and **CI/CD**.

9. Step completion checklist

Validation Item	Expected Result
.github/workflows/ci.yml present and functional	✅
Local npm run lint passes	✅
Local npm test -- --coverage passes (≥80%)	✅
GitHub Actions pipeline triggered automatically	✅
CI steps all green in Actions tab	✅
Coverage artifact downloadable	✅
Jira story shows build status = Passed	✅

At the end of **Step 3 — BUILD**, every feature branch should automatically:

- Run lint + tests + coverage checks

- Enforce minimum quality standards
- Report results to both GitHub and Jira

Step 4 — TEST (Vitest + Supertest)

This step focuses on verifying the quality and reliability of your application using **Vitest** (for unit testing) and **Supertest** (for integration testing).
You will run existing tests, understand their structure, and optionally add new ones to improve coverage.

Objective

- Ensure that all endpoints and utilities behave correctly.
- Maintain a **minimum coverage of 80%** across all source files.
- Automatically validate every Pull Request through the **CI workflow**.

1. Locate the test folders in your project

Open your ZIP project (Full_DevOps_Lab_Project_Final_Commented_EN.zip) and browse to:

```
test/
 ├── version.test.js
 ├── info.test.js
 ├── boom.test.js
 ├── health.test.js
 └── unit/
     ├── appInfo.test.js
     ├── status.test.js
     └── errorHandler.test.js
```

Folder Roles:

Folder	Purpose
test/	Integration tests → simulate real HTTP requests to the Express app
test/unit/	Unit tests → test isolated logic inside src/utils functions

2. Understand the test framework

The project uses:

- **Vitest** → testing framework compatible with Jest syntax
- **Supertest** → allows you to send fake HTTP requests to your Express app

Vitest configuration is already included in package.json:

```
"test": "vitest run",

"test:watch": "vitest",

"vitest": {

  "coverage": {

    "provider": "v8",

    "include": ["src/**/*.js"],

    "lines": 80,

    "functions": 80,

    "branches": 70,

    "statements": 80

  }

}
```

3. Run all tests locally

To execute all tests once:

```
npm test
```

To run continuously in watch mode (useful during development):

```
npm run test:watch
```

To include coverage analysis:

```
npm test -- --coverage
```

Expected console output:

```
✓ All tests passed

Coverage summary:

Statements   : 85%

Branches     : 72%

Functions    : 88%
```

```
Lines       : 84%
```

4. Integration tests (API endpoints)

Integration tests are located directly under test/.
They simulate real HTTP calls to your Express app through Supertest.

Example 1 — /version

File: test/version.test.js

```
describe("GET /version", () => {
  it("returns package version as a non-empty string", async () => {
    const res = await request(app).get("/version");
    expect(res.status).toBe(200);
    expect(typeof res.body.version).toBe("string");
  });
});
```

This test checks the /version route inside
src/routes/auto/version.route.js.

Example 2 — /info

File: test/info.test.js

```
describe("GET /info", () => {
  it("returns app info", async () => {
    const res = await request(app).get("/info");
    expect(res.status).toBe(200);
    expect(res.body).toHaveProperty("name");
    expect(res.body).toHaveProperty("uptime");
  });
});
```

This verifies the /info route in src/routes/auto/info.route.js, combining utilities from src/utils/appInfo.js.

Example 3 — /boom

File: test/boom.test.js

```
describe("GET /boom", () => {
  it("returns 500 with error payload", async () => {
    const res = await request(app).get("/boom");

    expect(res.status).toBe(500);

    expect(res.body.error).toBe(true);

  });

});
```

This ensures error handling works correctly with src/utils/errorHandler.js.

5. Unit tests (pure functions)

Unit tests are stored in test/unit/.
They directly import and test small reusable utilities.

Example 1 — formatStatus()

File: test/unit/status.test.js

```
describe("formatStatus", () => {

  it("returns 'steady' when uptime ≥ 3600s", () => {

    expect(formatStatus(3600)).toBe("steady");

  });

});
```

Tests the logic of src/utils/status.js ensuring 100% branch coverage.

Example 2 — getPackageInfo() and getRuntimeInfo()

File: test/unit/appInfo.test.js

```
describe("getPackageInfo", () => {

  it("returns name and version", () => {

    const info = getPackageInfo();

    expect(info).toHaveProperty("name");

    expect(info).toHaveProperty("version");

  });

});
```

Validates data reading from package.json inside src/utils/appInfo.js.

Example 3 — Error handler

File: test/unit/errorHandler.test.js

```
describe("errorHandler", () => {
  it("returns 500 with default message", () => {
    const res = mockRes();
    errorHandler({}, {}, res);
    expect(res.statusCode).toBe(500);
  });
});
```

Ensures proper handling and standardized error output across the app.

6. Add or extend tests (for higher coverage)

If you modify or add logic, you must also update or create corresponding test files.

Examples:

- Added a new condition in src/utils/status.js → Update test/unit/status.test.js
- New route /metrics → Create test/metrics.test.js using the same structure as others.

To check coverage visually:

```
open coverage/index.html
```

You'll see a color-coded breakdown (green = tested, red = missing).

7. Verify coverage in CI/CD

Every time you push your branch or open a Pull Request:

- GitHub Actions runs all tests automatically.
- If any test fails or coverage drops below thresholds, the pipeline will ❌ fail and block the merge.

In GitHub → **Actions tab → CI → Summary**, check:

- All tests passed
- Coverage artifact uploaded
- Download coverage-report.zip → open index.html

8. Troubleshooting

Problem	**Cause**	**Solution**
"app is not defined"	Missing import in test file	Add import app from "../src/app.js";
"Cannot read property version"	Wrong endpoint or missing field	Recheck the route response
Coverage below 80%	Missing branch or negative test	Add more unit cases

Problem	**Cause**	**Solution**
CI fails	Lint/test errors in code	Run npm run lint + npm test locally

9. Step Completion Checklist

Validation Item	**Expected Result**
All integration tests pass	✅
All unit tests pass	✅
Local coverage ≥ 80%	✅
CI pipeline test step green	✅
Coverage artifact available in GitHub Actions	✅
Jira story automatically updates build status	✅

End of Step 4 — TEST:
At this point, your project is fully verified — each student's feature is validated by both local tests and the CI pipeline.
Your Jira board should now show all stories in **"Review"** or **"Done"** with linked commits, branches, and build results.