# Lab 4 :

> Author: Badr TAJINI - DevOps Data for SWE - ESIEE - 2024/2025

---

# Lab: Version Control, Build Systems, and Automated Testing

## Introduction

In this lab, you will get hands-on experience with essential DevOps tools and practices, including version control with Git, collaboration using GitHub, setting up a build system with NPM, and writing automated tests using Jest and SuperTest. By the end of this lab, you will have a solid understanding of how to manage code collaboratively, automate common tasks, and ensure code correctness through testing.

## Objectives

- **Version Control with Git**
  - Initialize a Git repository
  - Commit changes and manage branches
  - Understand the basics of Git workflow
- **Collaborate Using GitHub**
  - Push local repositories to GitHub
  - Create and merge pull requests
  - Implement branch protection and code reviews
- **Build Systems with NPM**
  - Configure NPM scripts for common tasks
  - Manage dependencies using NPM
  - Build and run a Node.js application
- **Automated Testing**
  - Write and run tests using Jest and SuperTest
  - Understand different types of tests
  - Apply testing practices to application and infrastructure code

## Prerequisites

- Basic knowledge of the command line
- **Node.js** and **NPM** installed (Download Node.js)
- **Git** installed (Install Git)
- A **GitHub** account (Sign up for GitHub)
- **OpenTofu** installed (Install OpenTofu)
- **AWS** account with appropriate permissions
- Access to the Node.js sample app used in previous chapters

---

## Section 1: Version Control with Git

In this section, you will learn the basics of Git, a distributed version control system, and how to use it to track changes in your code.

### Steps

1. **Install Git**

   If you haven't already, install Git on your machine by following the instructions for your operating system:

   - Install Git

2. **Configure Git**

   Set your username and email in Git (replace `<YOUR NAME>` and `<YOUR EMAIL>` with your actual name and email):

   ```
   git config --global user.name "<YOUR NAME>"
   git config --global user.email "<YOUR EMAIL>"
   ```

3. **Initialize a Git Repository**

   Create a new directory for practicing Git and navigate into it:

   ```
   mkdir /tmp/git-practice
   cd /tmp/git-practice
   ```

Create a file and add some content:

```
echo 'Hello, World!' > example.txt
```

Initialize an empty Git repository:

```
git init
```

4. **Check Git Status**

   View the status of your repository:

   ```
   git status
   ```

   You should see that `example.txt` is an untracked file.

5. **Stage and Commit Changes**

   Add `example.txt` to the staging area:

   ```
   git add example.txt
   ```

   Commit the changes with a message:

   ```
   git commit -m "Initial commit"
   ```

6. **View Commit History**

   Check the commit history:

   ```
   git log
   ```

7. **Make Changes and Commit Again**

   Append a new line to `example.txt`:

   ```
   echo 'New line of text' >> example.txt
   ```

Check the differences:

```
git diff
```

Stage and commit the changes:

```
git add example.txt
git commit -m "Add another line to example.txt"
```

8. **Create and Switch Branches**

   Create a new branch called `testing` and switch to it:

   ```
   git checkout -b testing
   ```

   Modify `example.txt`:

   ```
   echo 'Third line of text' >> example.txt
   ```

   Stage and commit:

   ```
   git add example.txt
   git commit -m "Added a 3rd line to example.txt"
   ```

9. **Merge Branches**

   Switch back to the `main` branch:

   ```
   git checkout main
   ```

   Merge changes from `testing` into `main`:

   ```
   git merge testing
   ```

## Exercises

- **Exercise 1**: Use the `git tag` command to create a tag named `v1.0` on the current commit. Tags are useful for marking release points.

- ▶ Hint

- **Exercise 2**: Learn how to use `git rebase`. Create a new branch, make some commits, and then rebase it onto the `main` branch. Observe how `git rebase` differs from `git merge`.

  ▶ Hint

---

## Section 2: Collaborating with GitHub

Learn how to use GitHub to collaborate with others by pushing code to a remote repository and using pull requests.

### Steps

1. **Create a GitHub Account**

   If you don't have one already, sign up for GitHub.

2. **Create a New Repository**

   - Go to GitHub and click on "New" to create a new repository.
   - Name your repository (e.g., `devops-lab` or `other name`).
   - Choose "Public" or "Private".
   - Click "Create repository".

3. **Push Local Repository to GitHub**

   In your local repository, add the GitHub remote (replace `<YOUR_USERNAME>`):

   ```
   git remote add origin https://github.com/<YOUR_USERNAME>/devops-lab.git
   ```

   Push your commits to GitHub:

   ```
   git push -u origin main
   ```

4. **Pull Changes from GitHub**

   Make a change directly in GitHub (e.g., edit `example.txt` using the GitHub web interface).

   Pull the changes to your local repository:

```
git pull origin main
```

5. **Clone a Repository**

   If you didn't have a local repository, you could clone it using:

   ```
   git clone https://github.com/<YOUR_USERNAME>/devops-lab.git
   ```

6. **Create a Branch and Push to GitHub**

   Create a new branch:

   ```
   git checkout -b update-readme
   ```

   Create a `README.md` file and add some content:

   ```
   echo '# DevOps Lab' > README.md
   echo 'This is a sample repository for the DevOps lab.' >> README.md
   ```

   Commit and push:

   ```
   git add README.md
   git commit -m "Add README"
   git push origin update-readme
   ```

7. **Create a Pull Request**

   - Go to your repository on GitHub.
   - You should see a prompt to create a pull request for the `update-readme` branch.
   - Click "Compare & pull request".
   - Fill in the pull request details and submit.

8. **Review and Merge Pull Request**

   - Review the changes in the pull request.
   - Click "Merge pull request" and confirm.

9. **Pull Merged Changes Locally**

   Switch back to the `main` branch and pull the merged changes:

```
git checkout main
git pull origin main
```

### Exercises

- **Exercise 3**: Configure branch protection on your repository so that all changes to the `main` branch must be submitted via pull requests and approved by at least one reviewer.

  ▶ Hint

- **Exercise 4**: Enable signed commits on your repository to enhance security.

  ▶ Hint

---

## Section 3: Setting Up a Build System with NPM

In this section, you will set up NPM as a build system for a Node.js application, automating tasks like running the app and building Docker images.

### Steps

1. **Set Up the Directory Structure**

   Navigate to your main project directory and create a new directory for the sample app:

   ```
   cd ~/devops_base
   mkdir -p td4/scripts/sample-app
   cd td4/scripts/sample-app
   ```

2. **Copy the Sample App**

   Copy the `app.js` file from previous chapters or create a new one:

```
// app.js
const http = require('http');

const hostname = '127.0.0.1';
const port = 8080;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello, World!\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

3. **Initialize NPM**

   Initialize a new NPM project:

   ```
   npm init -y
   ```

   This will create a `package.json` file with default settings.

4. **Add a Start Script**

   Edit `package.json` to add a `start` script:

   ```
   "scripts": {
     "start": "node app.js"
   },
   ```

   Now you can run the app using:

   ```
   npm start
   ```

5. **Create a Dockerfile**

   Create a `Dockerfile` to containerize the app:

```
# Dockerfile
FROM node:21.7

WORKDIR /home/node/app

COPY package.json .
COPY app.js .

EXPOSE 8080

USER node

CMD ["npm", "start"]
```

6. **Write a Build Script**

   Create a script `build-docker-image.sh` to build the Docker image:

```
#!/usr/bin/env bash
set -e

name=$(npm pkg get name | tr -d '"')
version=$(npm pkg get version | tr -d '"')

docker buildx build \
  --platform=linux/amd64,linux/arm64 \
  --load \
  -t "$name:$version" \
  .
```

   Make it executable:

```
chmod u+x build-docker-image.sh
```

7. **Add a Dockerize Script**

   Edit `package.json` to add a `dockerize` script:

```
"scripts": {
  "start": "node app.js",
  "dockerize": "./build-docker-image.sh"
},
```

8. **Build the Docker Image**

Build the Docker image using NPM:

```
npm run dockerize
```

### Exercises

- **Exercise 5**: Modify the `Dockerfile` to use a specific Node.js version and explain why pinning versions can be important.

  ▶ Hint

- **Exercise 6**: Add a script to run your application inside a Docker container and document how to use it.

  ▶ Hint

---

## Section 4: Managing Dependencies with NPM

In this section, you will learn how to manage dependencies using NPM, adding external libraries to your project.

### Steps

1. **Install Express.js**

   Install Express.js as a dependency:

   ```
   npm install express --save
   ```

2. **Update `package.json`**

   Notice that `package.json` now includes Express in the `dependencies` section:

   ```
   "dependencies": {
     "express": "^4.19.2"
   },
   ```

3. **Rewrite `app.js` Using Express**

   Update `app.js` to use Express:

```
// app.js
const express = require('express');

const app = express();
const port = process.env.PORT || 8080;

app.get('/', (req, res) => {
  res.send('Hello, World!');
});

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

4. **Update the Dockerfile**

   Modify the `Dockerfile` to install dependencies:

```
# Dockerfile
FROM node:21.7

WORKDIR /home/node/app

COPY package.json .
COPY package-lock.json .

RUN npm ci --only=production

COPY *.js .

EXPOSE 8080

USER node

CMD ["npm", "start"]
```

5. **Rebuild the Docker Image**

   Rebuild the Docker image to include the new changes:

```
npm run dockerize
```

6. **Run the Application**

   Run the application to ensure everything works:

```
npm start
```

Access it at `http://localhost:8080`.

### Exercises

- **Exercise 7**: Add a new endpoint `/name/:name` that returns `Hello, <name>!` where `<name>` is a URL parameter.

  ▶ Hint

- **Exercise 8**: Explore how to add development dependencies ( `devDependencies` ) and explain the difference between `dependencies` and `devDependencies`.

  ▶ Hint

---

## Section 5: Automated Testing

Automated testing ensures that your application works as expected and helps catch bugs early in the development process. In this section, you will write automated tests for your Node.js application using Jest and SuperTest.

### Steps

1. **Install Testing Libraries**

   Navigate to your `sample-app` directory if you're not already there:

   ```
   cd ~/devops_base/td4/scripts/sample-app
   ```

   Install Jest and SuperTest as development dependencies:

   ```
   npm install --save-dev jest supertest
   ```

   This will add `jest` and `supertest` to your `devDependencies` in `package.json`.

2. **Update `package.json` Test Script**

   Modify the `test` script in your `package.json` to use Jest:

```
"scripts": {
  "start": "node app.js",
  "dockerize": "./build-docker-image.sh",
  "test": "jest --verbose"
},
```

3. **Refactor Your Application for Testing**

   Split your application code into separate files to make it easier to test.

   - **Update `app.js`**

     Modify `app.js` to export the Express app without starting the server:

     ```
     // app.js
     const express = require('express');

     const app = express();

     app.get('/', (req, res) => {
       res.send('Hello, World!');
     });

     app.get('/name/:name', (req, res) => {
       res.send(`Hello, ${req.params.name}!`);
     });

     module.exports = app;
     ```

   - **Create `server.js`**

     Create a new file `server.js` to start the server:

     ```
     // server.js
     const app = require('./app');

     const port = process.env.PORT || 8080;

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

   - **Update Start Script**

     Update the `start` script in `package.json`:

```
"scripts": {
  "start": "node server.js",
  "dockerize": "./build-docker-image.sh",
  "test": "jest --verbose"
},
```

4. **Write Tests**

   Create a test file `app.test.js`:

   ```
   touch app.test.js
   ```

   Add the following content to `app.test.js`:

   ```
   // app.test.js
   const request = require('supertest');
   const app = require('./app');

   describe('Test the root path', () => {
     test('It should respond to the GET method', async () => {
       const response = await request(app).get('/');
       expect(response.statusCode).toBe(200);
       expect(response.text).toBe('Hello, World!');
     });
   });

   describe('Test the /name/:name path', () => {
     test('It should respond with a personalized greeting', async () => {
       const name = 'Alice';
       const response = await request(app).get(`/name/${name}`);
       expect(response.statusCode).toBe(200);
       expect(response.text).toBe(`Hello, ${name}!`);
     });
   });
   ```

5. **Run Tests**

   Run your tests using NPM:

   ```
   npm test
   ```

   You should see output indicating that all tests have passed.

6. **Simulate a Bug**

Introduce a bug by changing the response in `app.js`:

```
app.get('/', (req, res) => {
  res.send('Hello, Mars!');
});
```

Re-run the tests:

```
npm test
```

Observe that the test for the root path now fails.

7. **Fix the Bug**

   Revert the change in `app.js` back to `'Hello, World!'` and ensure the tests pass again.

## Exercises

- **Exercise 9**: Add a new endpoint `/add/:a/:b` that returns the sum of two numbers. Write tests to validate both correct and incorrect inputs.

  ▶ Hint

- **Exercise 10**: Implement code coverage analysis using Jest and discuss the importance of code coverage in testing.

  ▶ Hint

---

## Section 6: Automated Testing for OpenTofu Code

In this section, you will add automated tests for your OpenTofu infrastructure code using OpenTofu's built-in testing capabilities.

### Steps

1. **Set Up Directory Structure**

   Navigate to your main project directory:

```
cd ~/devops_base
```

Create directories for OpenTofu tests:

```
mkdir -p td4/scripts/tofu/live
mkdir -p td4/scripts/tofu/modules/test-endpoint
```

2. **Copy the `lambda-sample` Module**

   If you have the `lambda-sample` module from previous chapters, copy it into the `td4/scripts/tofu/live` directory. If not, you can create a minimal module that deploys an AWS Lambda function and an API Gateway.

3. **Create the `test-endpoint` Module**

   Create a simple module in `td4/scripts/tofu/modules/test-endpoint` that can send HTTP requests to an endpoint.

   - **Example `main.tf`:**

     ```
     # main.tf in test-endpoint module
     data "http" "test_endpoint" {
       url = var.endpoint
     }

     variable "endpoint" {
       description = "The endpoint to test"
       type        = string
     }

     output "status_code" {
       value = data.http.test_endpoint.status_code
     }

     output "response_body" {
       value = data.http.test_endpoint.response_body
     }
     ```

4. **Write the Test Configuration**

   In the `lambda-sample` directory, create a file named `deploy.tftest.hcl`:

```
# deploy.tftest.hcl
run "deploy" {
  command = apply
}

run "validate" {
  command = apply

  module {
    source = "../../modules/test-endpoint"
  }

  variables {
    endpoint = run.deploy.api_endpoint
  }

  assert {
    condition     = data.http.test_endpoint.status_code == 200
    error_message = "Unexpected status code:
${data.http.test_endpoint.status_code}"
  }

  assert {
    condition     = data.http.test_endpoint.response_body == "Hello, World!"
    error_message = "Unexpected response body:
${data.http.test_endpoint.response_body}"
  }
}
```

**Note**: Ensure that `api_endpoint` is an output from your `lambda-sample` module that provides the URL to test.

5. **Run the Tests**

   Navigate to the `lambda-sample` directory and run the tests:

```
cd ~/devops_base/td4/scripts/tofu/live/lambda-sample
tofu test
```

   Wait for the tests to complete. They will deploy the infrastructure, validate it, and then destroy it.

6. **Verify Test Results**

   You should see output indicating whether the tests passed or failed.

### Exercises

- **Exercise 11**: Modify the test to check for a different response code or body content. For example, update your Lambda function to return a JSON response and adjust the test accordingly.

  ▶ Hint

- **Exercise 12**: Implement a negative test case where the endpoint should return an error code (e.g., 404) when accessing a non-existent resource.

  ▶ Hint

---

## Section 7: Testing Recommendations

In this section, we'll discuss best practices and recommendations for effective testing.

### Key Recommendations

1. **The Test Pyramid**
   - **Unit Tests**: Fast, isolated tests for individual components.
   - **Integration Tests**: Tests that combine components to ensure they work together.
   - **End-to-End Tests**: Tests that validate the entire application flow.
2. **What to Test**
   - Focus on critical functionality.
   - Test edge cases and error conditions.
   - Prioritize tests based on risk and impact.
3. **Test-Driven Development (TDD)**
   - Write tests before writing the implementation code.
   - Helps in defining clear requirements.
   - Encourages simpler and cleaner code.

### Exercises

- **Exercise 13**: Refactor one of your existing features using TDD. Write the test first, watch it fail, implement the feature, and then verify that the test passes.

▶ Hint

- **Exercise 14**: Analyze your test suite for code coverage and identify areas that lack sufficient tests.

  ▶ Hint

---

## Conclusion

In this lab, you have:

- **Explored Version Control**: Learned how to use Git for version control, manage branches, and collaborate using GitHub.
- **Set Up a Build System**: Configured NPM scripts to automate tasks like running the app and building Docker images.
- **Managed Dependencies**: Used NPM to manage application dependencies, ensuring consistent builds and deployments.
- **Implemented Automated Testing**: Wrote and ran automated tests for your Node.js application and OpenTofu infrastructure code.
- **Applied Testing Best Practices**: Discussed testing recommendations, including the test pyramid and TDD.

By following these practices, you can enhance code quality, facilitate team collaboration, and streamline your development workflow.

---

# Final Notes

Remember to commit your changes to your repository to keep track of your progress:

```
git add .
git commit -m "Complete lab on Version Control, Build Systems, and Automated Testing"
git push origin main
```

END