Lab (Option B: JavaScript): Using GitHub Actions - Part 2

Estimated time needed: 30 minutes

In this lab, you will build the Continuous Integration pipeline by setting up the steps under job in your workflow. Please ensure you complete setting up the workflow, before starting this lab.

You will add the following steps to the job in this lab:

- Check out the code
- Install dependencies
- Lint code with ESLint
- Run unit tests with Jest

Learning objectives

After completing this lab, you will be able to:

- Explain components of a job in a GitHub Actions workflow
- Use predefined GitHub actions
- Run multiline inline code in a step
- Explore action logs in the GitHub UI

Prerequisites

- It is highly recommended that you complete the lab titled "Using GitHub Actions - Setting up workflow." If you have not finished the lab, please finish that before starting this lab.
- A basic understanding of YAML.
- A GitHub account.
- An intermediate-level knowledge of CLIs.

Pre-work

1. Generate a personal access token on GitHub Settings, if you already don't have one.

2. Open a terminal window by using the menu in the editor: Terminal > New Terminal.

   First, let's run the following commands to install GitHub CLI.

   ```
   sudo apt update
   sudo apt install gh
   ```

3. Run the following command to authenticate with GitHub in the terminal with the personal access token.

   ```
   gh auth login
   ```

You will be taken through a guided experience, as shown here:

What account do you want to log into? **GitHub.com**
What is your preferred protocol for Git operations? **HTTPS**
Authenticate Git with your GitHub credentials. **Yes**
How would you like to authenticate GitHub CLI? **Paste an authentication token.**
Paste your authentication token: ************************************
You will be logged into GitHub as your account user.

4. Clone your copy of the forked GitHub repository. This creates the `ttwst-jhxyb-ci-cd-pipeline_js` subdirectory under `/home/project` directory.

   If you already have cloned this repository under /home/project, you can skip this step.

5. If you don't have `.github/workflows/workflow.yml` in `ttwst-jhxyb-ci-cd-pipeline_js` directory, create `workflow.yml` file in the `/home/project/ttwst-jhxyb-ci-cd-pipeline_js/.github/workflows` directory with the following code:

   ```
   name: CI workflow
   on:
     push:
       branches: [ "main" ]
     pull_request:
       branches: [ "main" ]
   jobs:
     build:
       runs-on: ubuntu-latest
       container: node:20-alpine
   ```

Step 1: Add the checkout step

You will now add your first step to the job. The first task is to check out the code from the repository. You will use the predefined `actions/checkout@v3` action to do this. You can learn more about this action on the GitHub page.

Your task

1. Add the `steps:` section at the end of the `build` job in the workflow file.

   ▼ Click here for a hint.

   ```
   steps:
   ```

2. Add the checkout step as the first step. Give it the `name: Check out` and call the action `actions/checkout@v3` by using the `uses:` keyword.

   ▼ Click here for a hint.

   ```
   steps:
     - name: {insert step name here}
       uses: {insert action name here}
   ```

Double-check that your work matches the solution below.

Solution

▼ Click here for the answer.
Replace the workflow.yml file with the code snippet below. You can also copy relevant parts of the code. Be sure to indent properly:

```
name: CI workflow
on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]
jobs:
  build:
    runs-on: ubuntu-latest
    container: node:20-alpine
    steps:
      - name: Checkout
        uses: actions/checkout@v3
```

Step 2: Install dependencies

Now that you have checked out the code, the next step is to install the dependencies. This application uses `npm`, the Node.js package manager, to install all the dependencies from the `package.json` file. The `node:20-alpine` container you are using already has the npm package manager installed.

The commands that you will use in this step are:

```
npm ci
```

This command installs all the dependencies listed in the package.json and package-lock.json files. The `ci` command, which stands for "clean install," is specifically designed for CI/CD pipelines. Unlike `npm install`, it does not modify the package-lock.json file and installs exactly what's specified there

Your task

1. Add a new named step after the `checkout` step. Name this step `Install dependencies`.

   ▼ Click here for a hint.

   ```
   - name: {insert step name here}
   ```

2. Next, you want to run the command to install the dependencies. You can run this using the `run:` keyword

   ▼ Click here for a hint.

   ```
   - name: Install dependencies
     run: |
       {insert code here}
   ```

Double-check that your work matches the solution below.

Solution

▼ Click here for the answer.
Replace the workflow.yml file with the code snippet below. You can also copy relevant parts of the code.

```
name: CI workflow
on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]
jobs:
  build:
    runs-on: ubuntu-latest
    container: node:20-alpine
    steps:
      - name: Checkout
        uses: actions/checkout@v3
      - name: Install dependencies
        run: |
          npm ci
```

Step 3: Lint with ESLint

It is always a good idea to add quality checks to your CI pipeline. This is especially true if you are working on an open source project with many different contributors. This makes sure that everyone who contributes is following the same style guidelines.

The next step is to use `ESLint` to lint the source code. Linting is checking your code for syntactical and stylistic issues. Some examples are spacing, using spaces or tabs for indentation, locating uninitialized or undefined variables, and missing brackets. The `eslint` library should be installed as a development dependency in the package.json file.

The ESLint command will analyze your JavaScript code according to the rules defined in your .eslintrc.* configuration file.

```
npm run lint
```

This command runs the lint script defined in your package.json file, which typically executes eslint on your source code.

Your task

1. Add a new named step after the `Install dependencies` step. Call this step `Lint with ESLint`.

   ▼ Click here for a hint.

   ```
   steps:
     - name: {insert step name here}
   ```

2. Next, you want to run the ESLint command to lint your code.

   ```
   npm run lint
   ```

   You can run inline commands using the `run` keyword with the pipe `|` operator.

   ▼ Click here for a hint.

   ```
   steps:
     - name: Lint with ESLint
       run: |
         {insert first command here}
   ```

Double-check that your work matches the solution below.

Solution

▼ Click here for the answer.
Replace the workflow.yml file with the code snippet below. You can also copy relevant parts of the code. Be sure to indent properly:

```
name: CI workflow
on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]
jobs:
  build:
    runs-on: ubuntu-latest
    container: node:20-alpine
    steps:
      - name: Checkout
        uses: actions/checkout@v3
      - name: Install dependencies
        run: |
          npm ci
      - name: Lint with ESLint
        run: |
          npm run lint
```

Step 4: Test code coverage with Jest

You will use `Jest` in this step to unit test the source code. Jest is a delightful JavaScript testing framework with a focus on simplicity. It works with projects using Babel, TypeScript, Node, React, Angular, Vue, and more.

Jest is configured via the jest.config.js file or in package.json to automatically collect code coverage information. At the end of the test run, you should see a percentage of coverage report.

Your task

1. Add a new named step after the `Lint with ESLint` step. Call this step `Run unit tests with Jest`.

   ▼ Click here for a hint.

   ```
   steps:
     - name: {insert step name here}
   ```

2. Next, you want to run the Jest command to test your code and report on code coverage.

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

This command runs Jest with the coverage flag enabled, which generates a code coverage report.

▼ Click here for a hint.

```
- name: Run unit tests with Jest
  run: {insert command here}
```

Double-check that your work matches the solution below.

Solution

▼ Click here for the answer.
Replace the workflow.yml file with the code snippet below. You can also copy relevant parts of the code. Be sure to indent properly:

```
name: CI workflow
on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]
jobs:
  build:
    runs-on: ubuntu-latest
    container: node:20-alpine
    steps:
      - name: Checkout
        uses: actions/checkout@v3
      - name: Install dependencies
        run: |
          npm ci
      - name: Lint with ESLint
        run: |
          npm run lint
      - name: Run unit tests with Jest
        run: npm test -- --coverage
```

Step 5: Push code to GitHub

To test the workflow and the CI pipeline, you need to commit the changes and push your branch back to the GitHub repository. As described earlier, each new push to the main branch should trigger the workflow.

Your task

1. Configure the Git account with your email and name using the `git config --global user.email` and `git config --global user.name` commands.

   ▼ Click here for a hint.

   Open the terminal and configure your email:

   ```
   git config --global user.email "you@example.com"
   ```

   Open the terminal and configure your user name

   ```
   git config --global user.name "Your Name"
   ```

2. The next step is to stage all the changes you made in the previous exercises and push them to your forked repo on GitHub.

   ▼ Click here for a hint.

   You can use the following commands to commit your changes to staging and then push to your forked repository:

   ```
   git add -A
   git commit -m "COMMIT MESSAGE"
   git push
   ```

Your output should look similar to the image below:

Solution

Step 6: Run the workflow

You can run the workflow now. Open the **Actions** tab in the forked repository in your GitHub account. Click the ***CI workflow*** action. Pushing changes in the previous step should have triggered a build action and it should be visible in the Actions tab.

Click the most recent `workflow run` to open the details page. You should see your workflow completed successfully:

You can now click the `build` job to see the logs for each step:

Spend some time here to expand each section:

If the workflow completes, you will see the `Complete job` step at the end of the build job logs.

Conclusion

Congratulations! You are now able to use GitHub Actions to create workflows for the CI pipeline.

In this lab, you added the steps to your Continuous Integration pipeline for your application. As a result, your application is automatically built and tested when you commit your code to the GitHub repository.

Next steps

Try to set up a GitHub action for your own project that checks out the code, runs unit tests, and performs linting on every push request to the default branch.

Author(s)

Tapas Mandal

Other Contributor(s)

Captain Fedora
John Rofrano