Lab: Add Continuous Integration

Estimated time needed: 60 minutes

A key practice in DevOps is continuous integration (CI), where developers continuously integrate their code into the main branch by making frequent pull requests. To assist them in this task, they use automation to confirm that every pull request runs the test suite to ensure that the proposed code changes will not break the build or reduce the test coverage.

Welcome to the **Add Continuous Integration** hands-on lab. In this lab, you will add a continuous integration workflow using GitHub Actions to run your tests with a PostgreSQL service. You will use the account service created in the previous lab to set up continuous integration automation using a CI pipeline. It is recommended that you complete the previous labs before beginning this one.

Objectives

In this lab, you will:

- Create a GitHub Actions workflow to run your CI pipeline
- Add events to trigger the workflow
- Add a job to the workflow
- Add steps to a job
- Run inline commands in the steps
- Review the logs for a workflow run
- View the activity for a workflow run

Note: Important Security Information

Welcome to the Cloud IDE with Docker. This is where all your development will take place. It has all the tools you will need to use Docker for deploying a PostgreSQL database.

It is important to understand that the lab environment is **ephemeral**. It only lives for a short while before it is destroyed. It is imperative that you push all changes to your own GitHub repository so that it can be recreated in a new lab environment any time it is needed.

Also note that this environment is shared and, therefore, not secure. You should not store any personal information, usernames, passwords, or access tokens in this environment for any purpose.

Finally, the environment may be recreated at any time, so you may find that you have to preform the **Initialize Development Environment** each time the environment is created.

Note on Screenshots

Throughout this lab, you will be prompted to take screenshots and save them on your device. These will be required either to answer graded quiz questions or for submission under **Option 1: AI-Graded Submission and Evaluation** or **Option 2: Peer-Graded Submission and Evaluation** at the end of this course. Your screenshot must have either the .jpeg or .png extension.

To take screenshots, you can use various free screen-capture tools or your operating system's shortcut keys. For example:

- **Mac:** you can use `Shift + Command + 3` (⇧ + ⌘ + 3) on your keyboard to capture your entire screen, or `Shift + Command + 4` (⇧ + ⌘ + 4) to capture a window or area. They will be saved as a file on your Desktop.

- **Windows:** you can capture your active window by pressing `Alt + Print Screen` on your keyboard. This command copies an image of your active window to the clipboard. Next, open an image editor, paste the image from your clipboard to the image editor, and save the image.

Initialize Development Environment

Because the Cloud IDE with Docker environment is ephemeral, it may be deleted at any time. The next time you come into the lab, a new environment may be created. Unfortunately, this means that you will need to initialize your development environment every time it is recreated. This should not happen too often as the environment can last for several days at a time but when it is removed; this is the procedure to recreate it.

Overview

Each time you need to set up your lab development environment, you will need to run three commands.

Each command will be explained in further detail, one at a time, in the following section.

`{your_github_account}` represents your GitHub account username.

The commands include:

```
git clone https://github.com/{your_github_account}/devops-capstone-project.git
cd devops-capstone-project
bash ./bin/setup.sh
exit
```

Now, let's discuss each of these commands and explain what needs to be done.

Task Details

Initialize your environment using the following steps:

1. Open a terminal with `Terminal` -> `New Terminal` if one is not open already.
2. Next, use the `export GITHUB_ACCOUNT=` command to export an environment variable that contains the name of your GitHub account.

   Note: Substitute your GitHub username for the `{your_github_account}` place holder below:

   ```
   export GITHUB_ACCOUNT={your_github_account}
   ```

3. Then, use the following commands to clone your repository, change into the `devops-capstone-project` directory, and execute the `./bin/setup.sh` command.

   ```
   git clone https://github.com/$GITHUB_ACCOUNT/devops-capstone-project.git
   cd devops-capstone-project
   bash ./bin/setup.sh
   ```

   You should see the following at the end of the setup execution:

4. Finally, use the `exit` command to close the current terminal. The environment will not be fully active until you open a new terminal in the next step.

   ```
   exit
   ```

Validate

In order to validate that your environment is working correctly, you must open a new terminal because the Python virtual environment will only activate when a new terminal is created. You should have ended the previous task by using the `exit` command to exit the terminal.

1. Open a terminal with `Terminal` -> `New Terminal` and check that everything worked correctly by using the `which python` command:

   Your prompt should look like this:

   Check which Python you are using:

   ```
   which python
   ```

   You should get back:

   Check the Python version:

   ```
   python --version
   ```

   You should get back some patch level of Python 3.9:

This completes the setup of the development environment. Anytime your environment is recreated, you will need to follow this procedure.

You are now ready to start working.

Exercise 1: Pick Up the First Story

The first thing you need to do is get a story to work on. You should never start coding without placing the story that you are working on into the **In Progress** column on the kanban board and assign it to yourself so that everyone knows you are working on it.

Your Task

1. Go to your kanban board and take the first story from the top of `Sprint Backlog`. It should be titled: "*Need the ability to automate continuous integration checks*".

2. Move the story to `In Progress`.

3. Open the story and assign it to *yourself*.

4. Read the contents of the story.

Results

The story should look like this:

Need the ability to automate continuous integration checks

As a Developer
I need automation to build and test every pull request
So that I do not have to rely on manual testing of each request, which is time-consuming

Assumptions

- GitHub Actions will be used for the automation workflow
- The workflow must include code linting and testing
- The Docker image should be postgres:alpine for the database
- A GitHub Actions badge should be added to the README.md to reflect the build status

Acceptance Criteria

```
Given code is ready to be merged
When a pull request is created
Then GitHub Actions should run linting and unit tests
And the badge should show that the build is passing
```

You are now ready to begin working on your story.

Exercise 2: Create a Workflow

The first thing you need to do is create a workflow for your GitHub Action. This should define the name and when to run the workflow.

Your Task

1. Take the first story "Need the ability to automate continuous integration checks" from `Sprint Backlog` and move it into `In Progress` and assign it to *yourself*.

2. Create a new branch named `add-ci-build` to work on in the development environment.

   ▼ Click here for a hint.

   ```
   git checkout -b {branch name here}
   ```

▼ Click here for the answer.

```
git checkout -b add-ci-build
```

3. Create a GitHub Actions workflow in the `.github/workflows` folder of your GitHub repository called `ci-build.yaml`

4. Give the workflow the name: `CI Build`.

▼ Click here for a hint.

```
name: {name here}
```

5. Set up the `on:` statement to run this workflow on every `push` and `pull_request` to the `main` branch of the repository.

▼ Click here for a hint.
Here is a template:

```
name: {name here}
on:
  push:
    branches:
      - {branch_name}
  pull_request:
    branches:
      - {branch_name}
```

Validate

Your workflow trigger definition should look like this:

▼ Click here to check your answer.

```
name: CI Build
on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main
```

You are now ready to create a job.

Exercise 3: Create a Job

Now that you know when to run the workflow, you need to create a job to contain your services and steps. You need to define a job runner, and you want this job to run inside a Python 3.9 Docker container.

Your Task

1. Create a job called `build`.

   ▼ Click here for a hint.
   Here is a template:

   ```
   jobs:
     build:
   ```

2. Specify that the build job runs on `ubuntu:latest`.

   ▼ Click here for a hint.
   Here is a template:

   ```
   jobs:
     build:
       runs-on: {runner_name}
   ```

3. Define a `container:` for the build job to run it that uses a `python:3.9-slim` image.

   ▼ Click here for a hint.
   Here is a template:

   ```
   jobs:
     build:
       runs-on: {runner_name}
       container: {image_name}
   ```

Validate

Your job section should look like this:

▼ Click here to check your answer.

```
jobs:
  build:
    runs-on: ubuntu-latest
    container: python:3.9-slim
```

You are now ready to create a database service.

Exercise 4: Define Required Services

Now that you have defined your job, you need to define any required services like databases. The account service uses a Postgres service to store its data. When running tests, it needs to have access to a Postgres database for testing. Luckily, GitHub Actions supports Docker containers, which allow you to start a database service in a Docker container, just like you do when you are developing.

Reference

This is the Docker command from your `Makefile` that is used for testing while developing locally. You can use the parameters from it to define a Postgres service for your workflow.

```
docker run -d --name postgres -p 5432:5432 \
        -e POSTGRES_PASSWORD=pgs3cr3t \
        -v postgres:/var/lib/postgresql/data \
        postgres:alpine
```

Your Task

1. Create a `services:` section in your workflow file at the same level of indentation as the `container:` section.

   ▼ Click here for a hint.
   Here is a template:

   ```
   jobs:
     build:
       runs-on: {runner_name}
       container: {image_name}
       services:
   ```

2. Under the `services:` section, define a service called `postgres:`.

   ▼ Click here for a hint.
   Here is a template:

   ```
   jobs:
     build:
   ```

```
    runs-on: {runner_name}
    container: {image_name}
    services:
      postgres:
```

3. Create an `image:` tag under the Postgres service with a value of `postgres:alpine`.

▼ Click here for a hint.
Here is a template:

```
jobs:
  build:
    runs-on: {runner_name}
    container: {image_name}
    services:
      postgres:
        image: {image name here...}
```

4. Create a `ports:` tag under the Postgres service that is listening on port `5432` (which is the default Postgres port).

▼ Click here for a hint.
Here is a template:

```
jobs:
  build:
    runs-on: {runner_name}
    container: {image_name}
    services:
      postgres:
        image: {image name here...}
        ports:
          - {port}:{port}
```

5. Add an `env:` environment variable tag and define values for `POSTGRES_PASSWORD` and `POSTGRES_DB`. Use the values `pgs3cr3t` and `testdb`, respectively.

▼ Click here for a hint.
Here is a template:

```
jobs:
  build:
    runs-on: {runner_name}
    container: {image_name}
    services:
      postgres:
        image: {image name here...}
        ports:
          - {port}:{port}
        env:
          POSTGRES_PASSWORD: {password here...}
          POSTGRES_DB: {database name here...}
```

6. Add an `options:` tag to check for a `health-cmd` of `pg_isready`, `health-interval` of `10s`, `health-timeout` of `5s`, and `health-retries` of `5`.

   ▼ Click here for a hint.
   Here is a template:

```
jobs:
  build:
    runs-on: {runner_name}
    container: {image_name}
    services:
      postgres:
        image: {image name here...}
        ports:
          - {port}:{port}
        env:
          POSTGRES_PASSWORD: {password here...}
          POSTGRES_DB: {database name here...}
        options: >-
          --health-cmd pg_isready
          --health-interval 10s
          --health-timeout 5s
          --health-retries 5
```

Validate

Your service definition should look like this:

▼ Click here to check your answer.

```
jobs:
  build:
    runs-on: ubuntu-latest
    container: python:3.9-slim
    services:
      postgres:
        image: postgres:alpine
        ports:
          - 5432:5432
        env:
          POSTGRES_PASSWORD: pgs3cr3t
          POSTGRES_DB: testdb
        options: >-
          --health-cmd pg_isready
          --health-interval 10s
          --health-timeout 5s
          --health-retries 5
```

You are now ready to start adding steps.

Exercise 5: Check Code and Install Dependencies

You are now ready to define the steps for your workflow. The first steps are to check out the code and install the dependencies. For checking out code, you will use an action from the GitHub Actions Marketplace.

To install the dependencies, you can use the same statements as if you were doing this locally on your computer.

Reference

You can use the following commands to install the Python package dependencies in your workflow steps.

```
python -m pip install --upgrade pip wheel
pip install -r requirements.txt
```

Your Task

1. Add a step named `Checkout` to check out the code using the `actions/checkout@v2` action.

▼ Click here for a hint.
Substitute the step name and action name from the instructions.

```
steps:
  - name: {checkout step name here...}
    uses: {action name here...}
```

2. Add a step named `Install dependencies` that upgrades `pip` and `wheel` and then uses the `pip` command to install the Python packages from the `requirements.txt` file.

▼ Click here for a hint.
Substitute the step name and command name from the instructions.

```
steps:
  - name: {checkout step name here...}
    uses: {action name here...}
  - name: {install step name here...}
    run: |
     {commands to install dependencies here...}
```

Validate

Your first two step definitions should look like this:

▼ Click here to check your answer.

```
jobs:
  build:
    services:
    # ...
    steps:
      - name: Checkout
        uses: actions/checkout@v2
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip wheel
          pip install -r requirements.txt
```

You are now ready to move on to the next step.

Exercise 6: Add Linting

In the **Assumptions** section of the story, it says:

- The workflow must include code linting and testing

In this step, you will add linting, which is checking your code for syntactical and stylistic issues. Some examples are line spacing, using spaces or tabs for indentation, locating uninitialized or undefined variables, and missing parentheses.

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 follows the same style guidelines.

Now, you will use `flake8` to lint the source code.

> Note: The flake8 library was installed as a dependency in the requirements.txt file.

Reference

You can use the following commands to run the `flake8` linter in your workflow steps.

```
flake8 service --count --select=E9,F63,F7,F82 --show-source --statistics
flake8 service --count --max-complexity=10 --max-line-length=127 --statistics
```

> Note: You should run these commands on your code before you add them to your workflow to ensure that your code passes the tests. You can use the `make lint` command to do this.

Your Task

1. Add a new step named `Lint with flake8` after the `Install dependencies` step that runs the `flake8` commands from the above reference.

▼ Click here for a hint.
Substitute the step name and command name from the instructions.

```
    steps:
      {... other steps here ...}
      - name: {step name here...}
```

```
          run: |
           {commands to run flake8 here...}
```

Validate

Your lint step definition should look like this:

▼ Click here to check your answer.

```
jobs:
  build:
    services:
    # ...
    steps:
      # ...previous steps here
      - name: Lint with flake8
        run: |
          flake8 service --count --select=E9,F63,F7,F82 --show-source --statistics
          flake8 service --count --max-complexity=10 --max-line-length=127 --statistics
```

You are now ready to move on to the next step.

Exercise 7: Add Unit Testing

To satisfy the testing requirements, you will use Nose in this step to unit test the source code. Nose is configured via the included `setup.cfg` file to automatically include the flags `--with-spec` and `--spec-color` so that red-green-refactor is meaningful. If you are in a command shell that supports colors, passing tests will be green and failing tests will be red.

Nose is also configured to automatically run the coverage tool, and you should see a percentage of coverage report at the end of your tests.

Reference

You can use the following commands to run the `nosetests` in your workflow steps.

```
nosetests -v --with-spec --spec-color --with-coverage --cover-package=service
```

> Note: You should run these commands on your code before you add them to your workflow to ensure that your code passes the tests. You can use the `make test` command to do this.

Your Task

1. Add a new step named `Run unit tests with Nose` after the `Lint with flake8` step that runs the `nosetests` commands from the above reference.

   ▼ Click here for a hint.
   Substitute the step name and command name from the instructions. Since you are running a single command, you do not have to use the pipe (|) operator with run.

   ```
   steps:
     {... other steps here ...}
     - name: {step name here...}
       run: {command to run nosetests here...}
   ```

2. Use the `env:` tag to add an environment variable named `DATABASE_URI` that will configure the tests to use the PostgreSQL database that you created in the `service:` section.

   ▼ Click here for a hint.
   Substitute the step name and command name from the instructions. Since you are running a single command, you do not have to use the pipe | operator with run.

   ```
   steps:
     {... other steps here ...}
     - name: {step name here...}
       run: {command to run nosetests here...}
       env:
         DATABASE_URI: "postgresql://postgres:{password}@{service}:{port}/{database}"
   ```

Validate

Your test step definition should look like this:

▼ Click here to check your answer.

```
jobs:
  build:
    services:
    # ...
    steps:
      # ...previous steps here
      - name: Run unit tests with nose
        run: nosetests
        env:
          DATABASE_URI: "postgresql://postgres:pgs3cr3t@postgres:5432/testdb"
```

That completes your workflow definition.

Check your final work

Try and run the workflow that you put together yourself. If anything goes wrong, here is the complete solution so that you can check how it may differ from yours.

▼ Click here to check the complete solution.

```
name: CI Build
on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main
jobs:
  build:
    runs-on: ubuntu-latest
    container: python:3.9-slim
    services:
      postgres:
        image: postgres:alpine
        ports:
          - 5432:5432
        env:
          POSTGRES_PASSWORD: pgs3cr3t
          POSTGRES_DB: testdb
        options: >-
          --health-cmd pg_isready
          --health-interval 10s
          --health-timeout 5s
          --health-retries 5
    steps:
      - name: Checkout
        uses: actions/checkout@v2
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip wheel
          pip install -r requirements.txt
      - name: Lint with flake8
        run: |
          flake8 service --count --select=E9,F63,F7,F82 --show-source --statistics
          flake8 service --count --max-complexity=10 --max-line-length=127 --statistics
      - name: Run unit tests with nose
        run: nosetests
        env:
          DATABASE_URI: "postgresql://postgres:pgs3cr3t@postgres:5432/testdb"
```

Exercise 8: Create a Badge

One of the last requirements listed in the story under **Assumptions** is:

- A GitHub Actions badge should be added README.md to reflect the build status

Now, it's time to add that badge.

The format of a GitHub Actions badge looks like this:

```
![Build Status](https://github.com/<OWNER>/<REPOSITORY>/actions/workflows/<WORKFLOW_FILE>/badge.svg)
```

- Where `<OWNER>` is the account name or organization name of the repository.
- `<REPOSITORY>` is the name of the repository, which you already know is `devops-capstone-project`.
- `<WORKFLOW_FILE>` is the name of the GitHub Actions workflow file that the badge represents, which you already know is `ci-build.yaml`.

Your Task

1. Edit the `README.md` file.
2. On a line below the title and separated from the title by a blank line above and below, add the following markdown:

```
![Build Status](https://github.com/<OWNER>/devops-capstone-project/actions/workflows/ci-build.yaml/badge.svg)
```

- Where `<OWNER>` is the name of your GitHub account.

3. Save the file and commit your changes using the message `Added badge for GitHub Actions`.

Once you make a pull request, the badge will show everyone the status of the build.

Exercise 9: Make a Pull Request

Now that you have completed your GitHub Action, you are ready to commit your changes, push code to your GitHub repository, and make a pull request.

Your Task

1. Commit your changes locally in the development environment with the message `"completed ci build"`.

▼ Click here for a hint.

```
git commit -am "{message here}"
```

▼ Click here for the answer.

```
git commit -am "completed ci build"
```

2. Push your local changes to a remote branch.

▼ Click here for the answer.

```
git push
```

3. Go to GitHub and make a pull request, which should kick off the GitHub Action that you just wrote.

Exercise 10: View the Workflow Run

When your workflow is triggered, a *workflow run* is created that executes the workflow. After a workflow run has started, you can see a visualization graph of the run's progress and view each step's activity on GitHub.

Your Task

1. On GitHub.com, navigate to the main page of the repository.
2. Under your repository name, click Actions.
3. In the left sidebar, click the workflow you want to see.
4. Under "Workflow runs," click the name of the run you want to see.
5. Under Jobs or in the visualization graph, click the job you want to see.
6. View the results of each step.
7. If everything worked, merge your pull request.

Evidence

For Option 1: AI-Graded Submission and Evaluation

1. Open the results of your GitHub Actions workflow run in the terminal.
   Copy the complete terminal output showing all workflow steps and save it in a file named **ci-workflow-done**.

▼ Click here for instruction to save terminal output

Follow the steps given below

1. Start by installing the GitHub CLI (`gh`) using the commands below:

   ```
   sudo apt update
   ```

   ```
   sudo apt install gh
   ```

2. Authenticate GitHub CLI using command `gh auth login`

Run the following command in your terminal:

```
gh auth login
```

When prompted, choose the options shown below:

1. What account do you want to log into?

   Select **GitHub.com**

2. What is your preferred protocol for Git operations?

   Select **HTTPS**

3. Authenticate Git with your GitHub credentials?

   Type **Y**

4. How would you like to authenticate GitHub CLI?

   Select **Paste an authentication token**

   > If you haven't already you can generate a Personal Access Token (PAT) by following the instructions given in the Hands-on Lab: Generate GitHub personal access token and be sure to include these scopes: `repo`, `read:org` and `workflow`

5. Paste your authentication token

   Paste your PAT into the terminal

   Note: If you see an `error validating token: missing required scope` error after entering your PAT, generate a new token and be sure to include these scopes: `repo`, `read:org` and `workflow`

Once done, you should see messages confirming:

- Git protocol configured
- Logged in with your GitHub username

3. Use the following commands to change directory:

   ```
   cd devops-capstone-project
   ```

4. To get the list of workflow runs for your GitHub repository run the following command from your project directory in the terminal:

   ```
   gh run list
   ```

After the list of workflow runs is displayed, pick the top most `run-id` from the output and view its details using command below:

```
gh run view <run-id> --verbose
```

Replace `<run-id>` with the ID shown in the list.

This will give you detailed information about that workflow run.

> Note: Ensure you are inside the correct repository directory and have successfully authenticated using `gh auth login` before running these commands.

Your output should appear similar to the image below:

5. Copy and paste the terminal output that shows your GitHub Actions workflow running successfully and save it in a text file named `ci-workflow-done` for the final project submission and evaluation. The output should clearly display the steps executed in the workflow.

2. If not done as part of question 1, Copy and paste the the public Github URL of `README.md` and save it in a text file that shows the updated badge.

3. Move the story **"Need the ability to automate continuous integration checks"** to the **Done** column on your Kanban board. Take a screenshot of your Kanban board and save it as `ci-kanban-done.jpeg` or `ci-kanban-done.png`.

4. Copy and paste the the public Github URL of `ci-build.yaml` and save it in a text file.

For Option 2: Peer-Graded Submission and Evaluation

1. Open the results of your workflow run on GitHub.
   Take a screenshot and save it as `ci-workflow-done.jpeg` or `ci-workflow-done.png`.
2. Go to your **README.md** file on GitHub.
   Take a screenshot of the README showing your workflow badge and save it as `ci-badge-done.jpeg` or `ci-badge-done.png`.
3. Move your story to the **Done** column on your Kanban board.
   Take a screenshot of your Kanban board and save it as `ci-kanban-done.jpeg` or `ci-kanban-done.png`.
4. Copy and paste the the public Github URL of `ci-build.yaml` and save it in a text file.

Conclusion

Congratulations! You have implemented the automation of unit testing during continuous integration using GitHub Actions. From this point on, you won't have to worry if the code you are merging will break the build.

Next Steps

In another lesson, you will implement the next story in Sprint 2 that will make more code changes and put your hard work creating a GitHub Action to good use.

Author(s)

Tapas Mandal
John J. Rofrano