CI/CD Pipeline with GitHub Actions and Heroku

Introduction

This project is a NodeJS-based web API, which uses a Postgres SQL database. The API quite minimal at this point, but allows a user to:

- Register as a new user (email and password) - returns a token if successful or an error if not
- Sign-in using a previously registered user account (email and password) - returns a token if successful or an error if not
- Retrieve their profile information (user ID and email) - by supplying the "token" issued when they registered or signed in

Forking the project

You can fork your own copy of the project from the repository at:

```
https://github.com/qa-apprenticeships/secure-api
```

Stage 1 - The build-and-test CI pipeline within GitHub

The original project repository has been set up with GitHub Actions to automate building and testing the project whenever code is checked in (i.e. it has a Continuous Integration CI pipeline within GitHub).

For security reasons, when you fork a repository, although this CI information is copied, it is initially disabled. You need to go through GitHub Actions within your fork and enable the actions.

The CI pipeline is actually defined by a YAML-format file called `node.js.yml` in the `.github/workflows` folder within the project. It's based on one that GitHub provided as a starter file, when I originally enabled GitHub actions and said the type of project was NodeJS.

The alterations to the yml file are mainly to give each step involved a label, and to always archive a copy of the test results, even if the test step itself failed (this would normally prevent subsequent steps from running, including he step that archives the test results). I do this because it's useful to archive failing tests as well as passing ones.

```yaml
name: Node.js CI
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2
    - uses: actions/setup-node@v1
      with:
        node-version: '12.x'
```

```
    - name: Set environment variables
      run: echo "NODE_ENV=test" >> $GITHUB_ENV
    - name: Prepare dependencies
      run: |
        npm ci
        npm run build --if-present
    - name: Run all tests
      run: npm test
    - name: Archive test and code-coverage results
      uses: actions/upload-artifact@v2
      if: ${{ always() }}
      with:
        name: test-and-code-coverage-report
        path: test-results.txt
```

Triggering the CI pipeline

To test the CI process within GitHub, simply make a change to one of the project files, such as adding a `//` comment to the start of the `src/app.js` file, then commit and push the change back to the repository. You should be able to see the CI process run within GitHub, monitor it, and review the log of what happened, including retrieving the generated test results.

Seeing the CI pipeline fail

To see what happens if a build or a test fails (i.e. when a developer breaks the build by mistake), make the following change...

- Edit the file `test/auth/01 register.spec.js`
- On line 52, change the start of the line from `it.skip('stores the password...` to `it('stores the password...` (i.e. remove the `.skip` part of the line)

This enables a previously skipped test, and it's a test that will fail because the code the feature doesn't work yet.

Commit the change, and push back to the repository. This should trigger the build, and the test stage should fail. You should be able to view the test results to see which test failed and why.

Don't forget to put the code back to how it was and push again, and confirm the pipeline now completes successfully.

Extra challenge...

See if you can follow how exactly the CI process works. Hints...

- Start with the `node.js.yml` file, look at the structure of it and the run commands in particular
- Look at the `package.json` file in the project, and look for entries in the `"scripts"` part of the file, where the name of the script item matches the `npm run X` line in the `node.js.yml` file, then look to see how each of the corresponding script items is defined, remembering that script actions sometimes invoke other script actions using `npm run X`
- Look at the `docker-compose.yml` file in the project - this is invoked by some of the script steps to set up temporary docker containers for the API and the database during the build/test process

Stage 2 - Deploying to Heroku

Heroku is a cloud-based hosting environment for lots of different types of projects, including NodeJS projects (like this one). We'll be setting up a deployment (CD) pipeline within Heroku so we can...

- Host the production version of the API and its database
- Host a staging version of the API and its database, for testing / demoing new versions prior to go-live
- Automatically deploy to staging whenever code in the main branch changes (but only if the CI pipeline within GitHub passes first)
- Promote the version currently in staging to production, just by clicking a button
- Host a 'preview' version of the API and its database each time a pull request is added to GitHub - so that proposed new features can be reviewed before deciding whether to accept them and merge them back into the main branch (at which point the feature automatically goes through to staging)

Note that each instance of the API (i.e. production, staging, and each feature preview) will have a different URL (and a separate database), so we can run all at the same time and can chose which one to interact with.

Prerequisites

Register for a free Heroku account, if you don't already have one. Then sign in.

Creating the pipeline

Within Heroku, say you want to add a new pipeline.

- Give the pipeline a name
- Connect the pipeline to your forked GitHub repository (click search and then connect)
- Go ahead and create the pipeline

Configure the Staging deployment

We want the application to be automatically deployed to Staging whenever changes are pushed to the main branch in GitHub. There are a few things we need to configure first to enable this.

- Within the Staging part of the pipeline, say you want to add an app
- Create a new app with these settings...
    - Name - relate it to the overall project, but probably include the word 'staging' somewhere
    - Region - Europe
- Click on the name of the staging app so we can edit its details
- On the app's Resources tab, do this:
    - Click configure add-ons
    - Add a Heroku Postgres add-on, under the 'Hobby - Dev' (free) pricing model
- On the app's Settings tab, do this:
    - Reveal the config vars
    - Add a key-value pair of `JWT_SECRET` = ABCDEFG1234567 (or any other random string)

    - Under the Buildpacks part, add a Buildpack of type NodeJS

- On the app's Deployment tab, do this:
    - Under the Automatic deploys part, confirm the deployment will be from the main branch
    - Tick the box to say wait for CI to pass before deploy
    - Click enable automatic deploys
    - Under the Manual deploy part, click Deploy branch to get our first staging deployment underway
- Assuming the deployment succeeded, open the app
- Test the app by navigating to /hi (within the app's main URL) - you should see a simple 'Hello' message

Configuring the Production deployment

We want to be able to promote the current Staging deployment through to Production, at the touch of a button. To set that up, do the following...

- Within the Production part of the pipeline, say you want to add an app
- Create a new app with these settings...
    - Name - relate it to the overall project, either with no mention of the environment, or by including the word 'prod' or 'production' somewhere
    - Region - Europe
- Click on the name of the production app so we can edit its details
- On the app's Resources tab, do this:
    - Click configure add-ons
    - Add a Heroku Postgres add-on, under the 'Hobby - Dev' (free) pricing model
- On the app's Settings tab, do this:
    - Reveal the config vars
    - Add a key-value pair of `JWT_SECRET` = `XYZXYZ123456` (or any other random string)
    - Under the Buildpacks part, add a Buildpack of type NodeJS
- Navigate back to the overall pipeline page
- Within the Staging part of the pipeline, click Promote to Production
- Assuming the deployment succeeded, open the production app
- Test the app by navigating to /hi (within the app's main URL) - you should see a simple 'Hello' message

Extra challenge...

Simulate changes being made to the main branch of the project, see the changes appear automatically in Staging, then promote the changes to Production when you are happy with them.

- Make a change to the message displayed when the /hi API is called (see the file `src/app.js`, around line number 22).
- Push your change into GitHub (main branch)
- Monitor the automatic deployment to Staging

- Confirm that the Staging app has been updated
- Confirm that the Production app remains unchanged from before
- Promote Staging to Production
- Confirm that Production has now been updated

Configuring pull-request preview deployments

We'd like to have the pipeline automatically spin up a new environment every time we add a pull request on a feature branch to GitHub. This will allow us to easily try out the new feature or change added by the pull request, before deciding whether to accept the pull request (at which point the change will be merged into the main branch, and will be automatically deployed to Staging).

- In the overall pipeline page, click Enable Review Apps, then...
    - Tick the box labelled Create new review apps for new pull requests automatically
    - Tick the box labelled Wait for CI to pass
    - Tick the box labelled Destroy stale review apps automatically (after 5 days)
    - Pick region: Europe
    - Click Enable review apps
- Back on the overall pipeline page, select the Settings tab (so we can configure some settings for the "template" used to create each review app), then...
    - Scroll down to the Review apps section
    - Click Reveal config vars
    - Add a key-value pair of `JWT_SECRET` = XYZXYZ123456 (or any other random string)

To test the review apps work, simulate a pull request for a new feature by doing the following...

- Create a new branch called 'feature-1' (or whatever), and switch to it
- Make a simple change, such as changing the /hi message (see above)
- Commit and push the change to the corresponding feature branch in GitHub
- In GitHub, create a new pull request connecting your feature branch to the main branch
- You should see a new review app get created in Heroku specifically for this pull request (it takes a few moments, because the CI pipeline must first complete in GitHub)
- Try opening the review app, and check the /hi API returns the message defined by the feature branch
- Verify that the app currently deployed to Staging has not been affected
- In GitHub, look at the pull request - you should see information integrated from Heroku about the new environment(s) created for this pull request
- In GitHub, accept the pull request by merging the feature branch into the main branch
- This should trigger the feature to be deployed to Staging - check this has happened too
- Verify that the review app for the (now closed) pull request has automatically been destroyed

Extra challenge...

Research how these files support the Heroku CD pipeline - they were added as part of the project's development, but are specific to the Heroku CD pipeline.

- app.json (hint: needed for "review" apps)
- Procfile (hint: how does Heroku know what to do after deploying the code to an environment, to get the environment's database scheme initialised / migrated, how/when does it make sense to seed the database with some sample data, plus how does it start the NodeJS API seb server?)

Stage 3 - Be the Developer!

Work with this project from the developer's point of view. Here are some pointers...

- Pre-requisites...
    - Docker (desktop) installed on your development machine
    - For Windows machines, Docker configured to use WSL2, with Ubuntu installed
    - NodeJS 14.x (and npm) installed on your development machine (within WSL for Windows - ask for help!)
- Clone the repository onto your development machine (Windows or Mac)
- If running on Windows, make sure to tell VS Code to open the folder in WSL
- Run the command `npm install` (to download all dependencies of the project)
- Run the command `docker-compose up -d` (to spin up Docker container for the development and integration testing databases, and also a container for `adminer` - so you can interactively connect to one of the databases and poke around in its data if you need to)
- Alternatively, you can spin up the containers using the commands `npm run start-db` (to start the database) and `npm run adminer` (to start adminer)
- When you've finished developing, you can shut down the no-longer-needed Docker containers using either `docker-compose down` or `npm run down`
- To run the development server, so it serves up the API, use the command `npm run dev`
- To kill the development server, use `ctrl-C`
- When running, the API will be served on the URL `http://localhost:8001`
- See the file `http-demos/auth.http` for details of all the API methods available, and how to call them (note that some of the API methods require a POST request, which you can't do direct through a browser - either use the REST Client extension for Visual Studio Code, or use a product such as Postman)
- To run all the automated tests (including coverage analysis which must be 100%), use the command `npm run test`