Designing and Implementing a Backend API Structure:

Development Report

Stewart Charles Fisher II
Student ID: 25020928
QLS ID: FIS20772963
25020928@students.lincoln.ac.uk

April 2024

School of Computer Science
University of Lincoln
United Kingdom

Submitted in partial fulfilment of the requirements for the Degree of MComp Computer Science

Word Count: 1053

Contents

Chapter 1

Source Code Control

The project has been managed via a Git repository, initialised on the 8th of February, 2024. Using Git allows me to make checkpoints of progress during the development process, tracking the changes made to the codebase. If a change caused a breaking error, the repository enabled me to revert back to a prior commit. In the industry, Git is crucial as it facilitates a team's ability to productively work on multiple platforms simultaneously.

1.1 Branching Workflow

The development flow structure chosen for the repository was to have a main branch dedicated to packaging feature-complete releases, a development branch dedicated to testing new features, and feature branches to develop new features on.

1.2 GitHub

GitHub was the remote repository platform used to host the project, owing to the features that it provides:

Issue tracking: GitHub allowed me to create issues for particular ideas or tasks that could or should be addressed, along with dedicated branches.

CI/CD: GitHub allowed me automate my development process, particularly testing and deployment, using GitHub Actions and custom workflows.

It is important to note that GitHub was not the only option for a remote repository host in this project. GitLab essentially offers the same functionality as GitHub with regards to the basics of version control, however it provides functionalities that are superior in a large scale enterprise operation:

DevOps focus: GitLab tailors its toolset towards developers focused on the software development process, whereas GitHub is tailored towards developers who prioritise a collaborative development environment.

CI/CD integration: GitHub allows the user to pick the CI/CD toolset after the project is integrated, whereas GitLab has a CI/CD toolset integrated into the platform itself.

Self hosting: Enterprises tend to favour self-hosting their version control environments for improved control over their development process; GitLab has better support for self hosting.

1.3 Poetry

Poetry is a Python package management tool, similar to Pip. While Pip is easier to interpret, Poetry provides a better functionality for package version control, it has a improved sub-dependency resolver, and has built-in support for virtual environments. For a large scale project, as this project intends to be, version control of the packages within the artefact is as important as version control of the artefact itself; using Pip along with a `requirements.txt` file is not strict enough to maintain this. Furthermore, if a dependency has to be removed, Pip is not capable of removing sub-dependencies like Poetry, meaning that useless packages could accrue a large amount of wasted space on a deployment server.

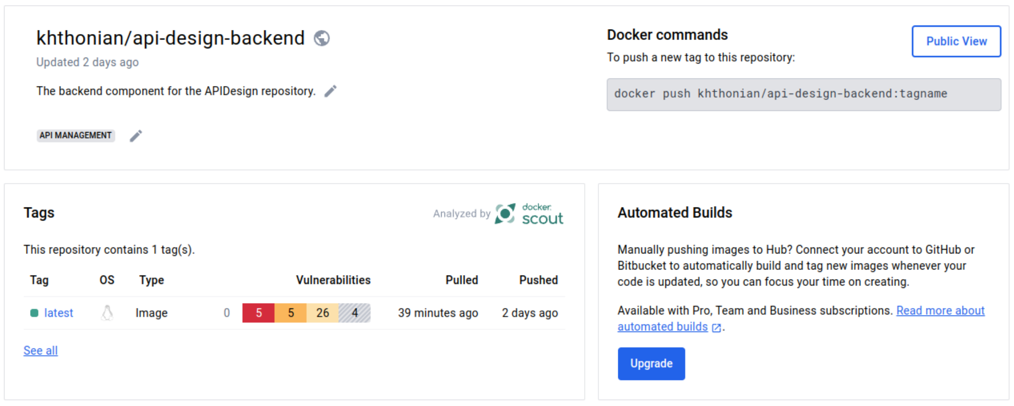
1.4 Alembic

In order to ensure that the SQLite database correctly interacts with the SQLAlchemy models, the Alembic database migration tool was used. Alembic, written by the same author as SQLAlchemy, allowed me to manage changes to the models using a version control management system that is similar in function to Git; a change could be made and applied to the `weather.db` database file that, in the case it didn't work, could be reverted to a prior version.

1.5 Docker Hub

As the artefact uses Docker containers for the backend and the frontend, the container images were stored to the Docker Hub platform so that the built images could be version controlled alongside the source code. In an enterprise environment, using a container registry, such as Docker Hub, significantly reduces the time that would be spent rebuilding the images across the server machines.

Alongside this, the Docker Scout analysis service freely checks the images for security vulnerabilities, reducing the human error factor of only checking for vulnerabilities using unit testing; see Figure 1.1.



khthonian/api-design-backend

Updated 2 days ago

The backend component for the APIDesign repository.

API MANAGEMENT

Docker commands

To push a new tag to this repository:

```
docker push khthonian/api-design-backend:tagname
```

Public View

Tags

Analyzed by docker scout

This repository contains 1 tag(s).

Tag	OS	Type	Vulnerabilities					Pulled	Pushed
latest		Image	0	5	5	26	4	39 minutes ago	2 days ago

See all

Automated Builds

Manually pushing images to Hub? Connect your account to GitHub or Bitbucket to automatically build and tag new images whenever your code is updated, so you can focus your time on creating.

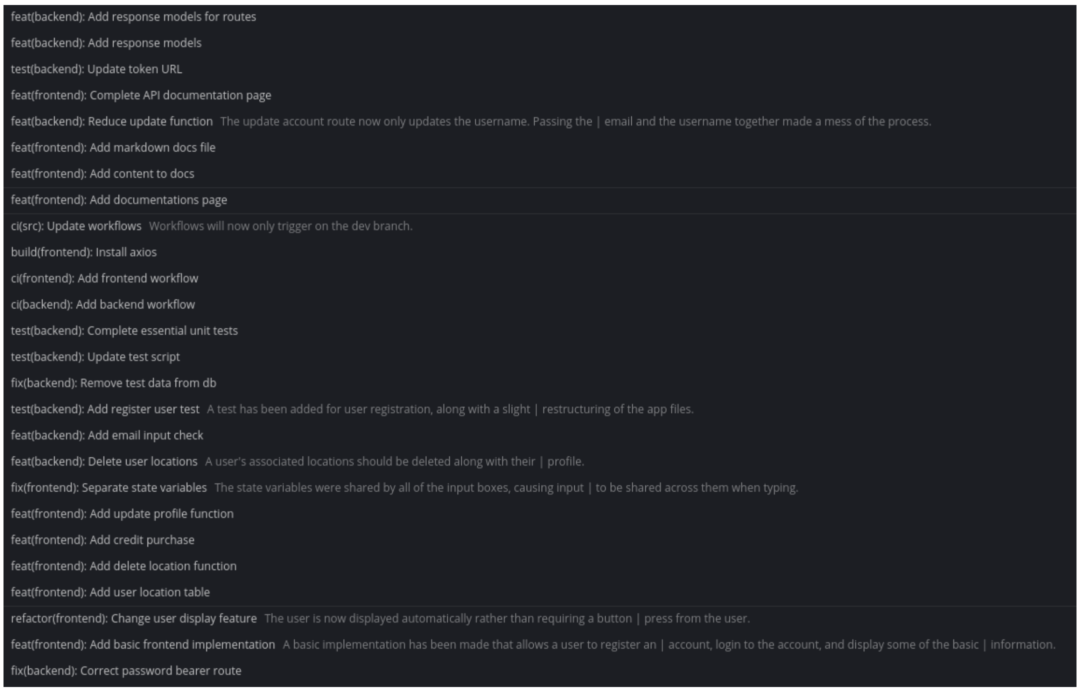
Available with Pro, Team and Business subscriptions. Read more about automated builds.

Upgrade

Figure 1.1: An example of the Docker Hub platform

1.6 Commit Styling

A minor facet to the project was to ensure that correct formatting was used for the commit history. Given the scale of the project and large number of commits, it was important that commits could easily be filtered and found. To do this, the Conventional Commits protocol was followed; see Figure 1.2.



```
feat(backend): Add response models for routes
feat(backend): Add response models
test(backend): Update token URL
feat(frontend): Complete API documentation page
feat(backend): Reduce update function  The update account route now only updates the username. Passing the | email and the username together made a mess of the process.
feat(frontend): Add markdown docs file
feat(frontend): Add content to docs
feat(frontend): Add documentations page
ci(src): Update workflows  Workflows will now only trigger on the dev branch.
build(frontend): Install axios
ci(frontend): Add frontend workflow
ci(backend): Add backend workflow
test(backend): Complete essential unit tests
test(backend): Update test script
fix(backend): Remove test data from db
test(backend): Add register user test  A test has been added for user registration, along with a slight | restructuring of the app files.
feat(backend): Add email input check
feat(backend): Delete user locations  A user's associated locations should be deleted along with their | profile.
fix(frontend): Separate state variables  The state variables were shared by all of the input boxes, causing input | to be shared across them when typing.
feat(frontend): Add update profile function
feat(frontend): Add credit purchase
feat(frontend): Add delete location function
feat(frontend): Add user location table
refactor(frontend): Change user display feature  The user is now displayed automatically rather than requiring a button | press from the user.
feat(frontend): Add basic frontend implementation  A basic implementation has been made that allows a user to register an | account, login to the account, and display some of the basic | information.
fix(backend): Correct password bearer route
```

Figure 1.2: An excerpt from the Git commit history

Chapter 2

Testing Protocol

The testing approach used for the project was white box testing, specifically unit testing, integration testing, and regression testing; unit testing ensures that each component works as expected, integration testing ensures that the components of the API, primarily the access token endpoint, work correctly in combination, and regression testing ensures that changes to the codebase don't cause unintentional changes to the execution.

Both happy path and unhappy path unit tests were implemented for the endpoints, to validate for both expected outputs and error handling in invalid circumstances. This was necessary to ensure that the intended operation was successful but also to try and protect against instances where users could intentionally or unintentionally use the endpoints in a fashion that could cause an issue to the user or the system at large.

Due to my API using a SQLite database, it was crucial that the unit test cases were tested using a SQLite database rather than a fake database via a Python dictionary. If I had only used a Python dictionary, it would mean that I wouldn't have been testing the SQLAlchemy models that had been defined, and applied to the database via Alembic, meaning an error could potentially occur in a production phase. The database for testing was initialised to run in memory, so that it didn't interact with the dedicated `weather.db` database file that is passed to the Docker container.

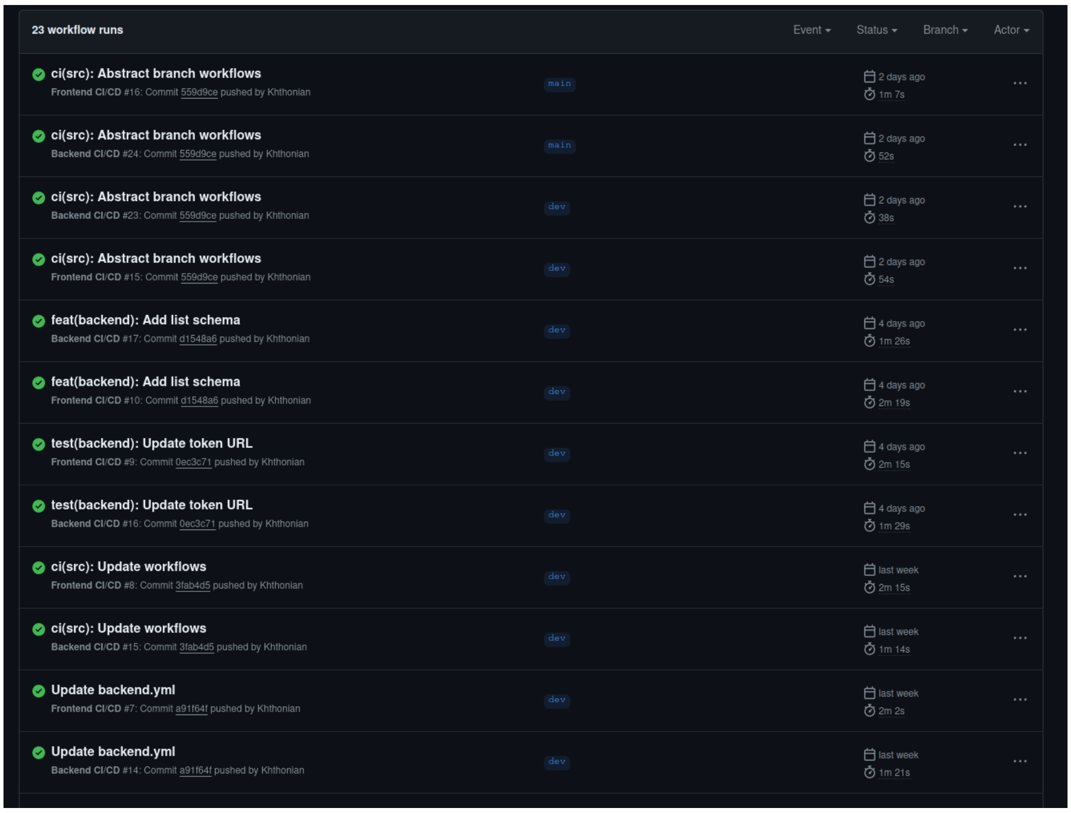
Chapter 3

CI/CD

GitHub Actions were used to automate the testing and deployment functions for the project; see Figure 3.1. Two workflow files were made, for the backend and the frontend, both containing two jobs:

Build: A job to build the components and run the unit tests[1] to ensure operational integrity.

Deploy: A job to build the Docker container images and push them to Docker Hub.



23 workflow runns

Event ▾ Status ▾ Branch ▾ Actor ▾

✓ **ci(src): Abstract branch workflows** **Frontend CI/CD** #16: Commit 559d9ce pushed by Khthonian	main	2 days ago 1m 7s	...
✓ **ci(src): Abstract branch workflows** **Backend CI/CD** #24: Commit 559d9ce pushed by Khthonian	main	2 days ago 52s	...
✓ **ci(src): Abstract branch workflows** **Backend CI/CD** #23: Commit 559d9ce pushed by Khthonian	dev	2 days ago 38s	...
✓ **ci(src): Abstract branch workflows** **Frontend CI/CD** #15: Commit 559d9ce pushed by Khthonian	dev	2 days ago 54s	...
✓ **feat(backend): Add list schema** **Backend CI/CD** #17: Commit d1548a6 pushed by Khthonian	dev	4 days ago 1m 26s	...
✓ **feat(backend): Add list schema** **Frontend CI/CD** #10: Commit d1548a6 pushed by Khthonian	dev	4 days ago 2m 19s	...
✓ **test(backend): Update token URL** **Frontend CI/CD** #9: Commit 0ec3c71 pushed by Khthonian	dev	4 days ago 2m 15s	...
✓ **test(backend): Update token URL** **Backend CI/CD** #16: Commit 0ec3c71 pushed by Khthonian	dev	4 days ago 1m 29s	...
✓ **ci(src): Update workflows** **Frontend CI/CD** #8: Commit 3fab4d5 pushed by Khthonian	dev	last week 2m 15s	...
✓ **ci(src): Update workflows** **Backend CI/CD** #15: Commit 3fab4d5 pushed by Khthonian	dev	last week 1m 14s	...
✓ **Update backend.yml** **Frontend CI/CD** #7: Commit a91f64f pushed by Khthonian	dev	last week 2m 2s	...
✓ **Update backend.yml** **Backend CI/CD** #14: Commit a91f64f pushed by Khthonian	dev	last week 1m 21s	...

Figure 3.1: An example of workflows used in the project

Caches were used for the workflows, to reduce the time that could be spent by continually reinstalling packages to the runner.

[1]Testing was exclusive to the backend.

3.1 Branch Abstraction

Even though there was one workflow for the frontend and one workflow flow for the backend, each workflow script covered both the main and development branch. The build workflow job would run on the development branch, while the deploy workflow job would run on the main branch. To do this, conditionals were applied to the jobs; see Listing 3.1.

```
jobs:
  build:
    if: github.ref == 'refs/heads/dev'
    runs-on: ubuntu-latest
...
  deploy:
    if: github.ref == 'refs/heads/main'
    runs-on: ubuntu-latest
```

Listing 3.1: The branch conditionals for the workflow jobs

3.2 Repository Secrets

Due to the unit tests requiring the API keys, and the Docker Hub integration requiring my Docker Hub login credentials, GitHub secrets had to be used; see Listing 3.2. GitHub secrets allow you to store sensitive environment information to the repository so that they can be used in workflows without exposing them.

```
env:
  PYTHON_VERSION: "3.11"
  POETRY_VERSION: "1.8.2"
  ALGORITHM: ${{ secrets.ALGORITHM }}
  GEOLOCATION_API_KEY: ${{ secrets.GEOLOCATION_API_KEY }}
  OPENAI_API_KEY: ${{ secrets.OPENAI_API_KEY }}
  OPENWEATHER_API_KEY: ${{ secrets.OPENWEATHER_API_KEY }}
  SECRET_KEY: ${{ secrets.SECRET_KEY }}
...
- name: Login to Docker Hub
  uses: docker/login-action@v3
  with:
    username: ${{ secrets.DOCKER_USERNAME }}
    password: ${{ secrets.DOCKERHUB_TOKEN }}
```

Listing 3.2: The environment secrets in the workflow jobs