COMP0034 Coursework 1

In this coursework, I created a RESTAPI that uses Flask and the Estates Management Record from HESA (see references). To run the app:

1. Fork this repository: **https://github.com/ucl-comp0035/comp0034-cw1i-4jjnaomi**
2. Clone the resulting repository locally and to your IDE
3. Create and activate a virtual environment
4. Install the requirements using pip install -r requirements.txt
5. Run the app flask --app src run --debug
6. Open a browser and go to http://127.0.0.1:5000/
7. Go to the various URLs outlined in List of URLs
8. Stop the app using CTRL+C

Application code

The code that creates the REST API is found in the src directory of the project. This Flask app can be run in development mode from the command line of a terminal. The URLs for each route can be seen in Table 1.

List of URLs

Table 1 Available urls for flask app and explantion of the REST API methods associated

Method	URL	Explanation	Example JSON input
N/A	/	Index page	N/A
GET	/hei?page=1&per_page=10	Obtains all hei entries in the database. The 'page' and per_page' variables can be specified by the user	N/A
GET	/hei/<ukprn>	Obtains a specific hei using its UKPRN. /hei/10006840 will obtain the data for The University of Birmingham	N/A
POST	/hei	Adds a new HEI to the database	{ "UKPRN": "12222221", "he_name": "New Random University", "region": "Random Region", "lat": "", "lon": "" }
DELETE	/hei/<ukprn>	Deletes a HEI from database using the provided UKPRN in the URL. /hei/10006840 will delete the data for The University of Birmingham	N/A

PUT	/hei/<ukprn>	Updates an HEI if the ukprn exists or adds a new UKPRN if it doesn't already exist. Requires all the fields of the HEI model	{
			"UKPRN": "12222222",
			"he_name": "New Random University",
			"region": "Random Region",
			"lat": "",
			"lon": ""
			}

PATCH	/hei/<ukprn>	Updates an HEI that already exist in the database. Doesn't require all fields of the model.	{
			"lat": "111",
			"lon": "111"
			}

GET	/entry?page=1&per_page=10	Obtains all entries in the database. The 'page' and per_page' variables can be specified by the user	N/A
GET	/entry/<id1>	Obtains a specific entry using its id /entry/1 will obtain the data for the first entry	N/A

POST	/entry	Adds a new entry to the database	{
			"entry_id": "100000",
			"academic_year": "20/20",
			"classification": "dummy",
			
			"category_marker": "dummy",
			"category": "dummy",
			"value": "70",
			"UKPRN": "111111",
			"he_name": "University of Naomi"
			}

DELETE	/entry/id1	Deletes an entry from database using the provided id in the URL. /entry/1 will delete the data for the first entry	N/A
PUT	/entry/id1	Updates an entry if the id exists or adds a new id if it doesn't already exist. Requires all the fields of the entry model	{ "entry_id": "100001", "academic_year": "20/20", "classification": "dummy", "category_marker": "dummy",

			"category": "dummy", "value": "70", "UKPRN": "111111", "he_name": "University of Naomi" }

PATCH	/entry/id1	Updates an entry that already exists in the database. Doesn't require all fields of the model.	{
			"value": "100000",
			}

Pagination

As can be seen in the table above, the HEI and Entry GET routes have pagination. This is because both of these data tables have more than 100 records. This therefore gives the user the ability to customise the view obtained in the app.

Test code

Evidence of tests

As shown in the Figure 1, there were 34 tests and they all passed. The tests can be seen in the tests directory. The tests use parameterisation to access multiple route endpoints. The tests also use patching to mock certain database operations and simulate exceptions for error handling scenarios.

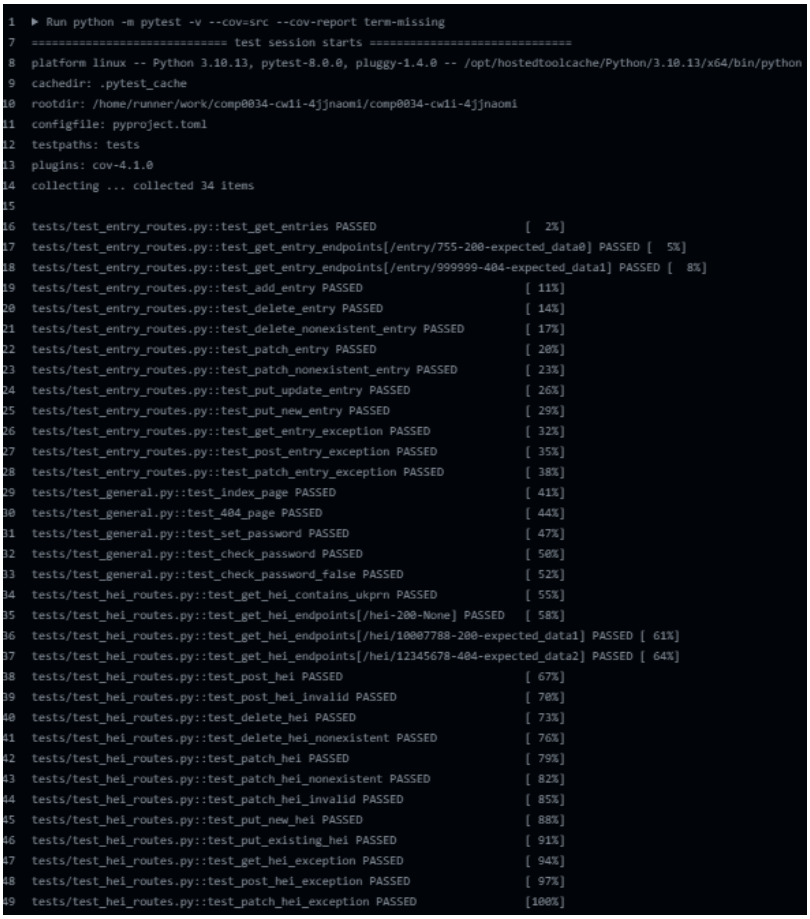

Figure 1 Evidence of tests being run for REST API app

Coverage

Figure 2 shows the output from the pytest coverage. This shows that I achieved 86% coverage which is good for an app with this amount of code. I looked at the areas where I was missing tests, and this majorly came from the exception block for Marshmallow ValidationErrors. I attempted to write a test this that mocked the ValidationError but obtained the following error: `sqlite3.OperationalError: database table is locked`. ChatGPT told me that this error often arises when using SQLite in testing environments where multiple tests are running concurrently, attempting to access the same database file. To address the error, I could have used an in-memory SQLite database for testing which would've avoided file

locking issues. However, this would require a lot more time and knowledge than what is needed for this course.

```
---------- coverage: platform linux, python 3.10.13-final-0 ----------
Name                    Stmts   Miss Branch BrPart  Cover   Missing
-------------------------------------------------------------------
src/__init__.py            61      5     18      4    89%   59->71, 72->exit, 103, 109-110, 142-143
src/controllers.py        194     16     20      4    91%   66-71, 186->190, 196->204, 250-253, 304-307, 356->360, 365->372, 368-371
src/error_handlers.py      21     21      2      0     0%   4-74
src/models.py              43      0      0      0   100%
src/schemas.py             16      0      0      0   100%
-------------------------------------------------------------------
TOTAL                     335     42     40      8    86%


============================ 34 passed in 14.72s =============================
```

Figure 2 Coverage report for tests run for REST API app

The coverage report also showed that I was missing tests for my add_data_from_csv function in the directory initialiser. Adding tests for this woud be difficult as I would need to overwrite my csv files which would cause issues when running the app while not in testing mode.

Tools and techniques

Github repository: https://github.com/ucl-comp0035/comp0034-cw1i-4jjnaomi

Continuous integration

This development of this app used a GitHub Actions workflow so that testing was done whenever changes were made to the repository. This meant that I was able to fix issues quickly. Evidence of the workflow being used is shown in Figure 3.

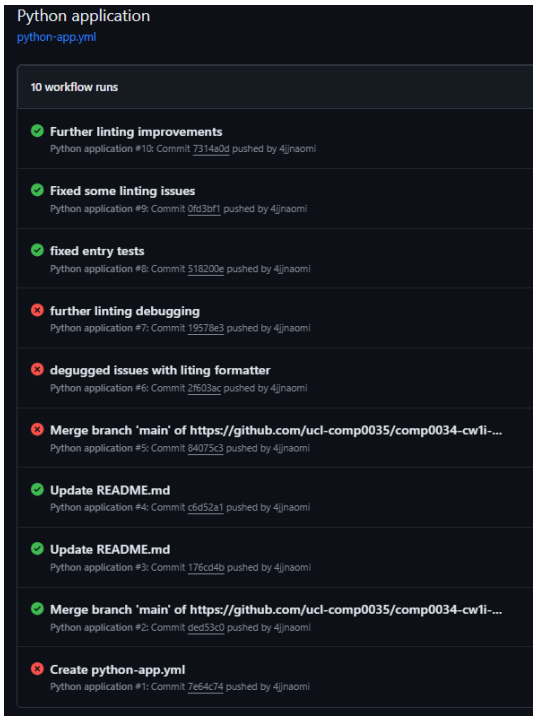

Figure 3 Evidence of GitHub Actions workflow being used

Linting

In writing the code, a linter was used within Visual Studio Code to ensure good code quality through the identification and reporting of potential errors. Figure 4 shows some of the issues highlighted by pylint with my code.

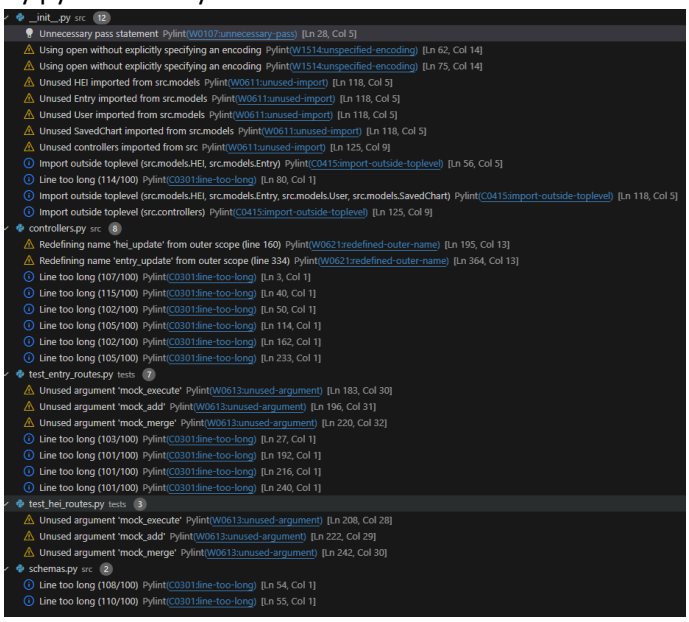

Figure 4 Evidence of pylint output in VSCode

Although pylint in VSCode is useful, some of the errors highlighted are irrelevant. For example, the 'uncessary pass statement' problem was ignored because the use of pass in the block of code stated is required for the logic of the app and to ensure the app didn't break.

Therefore, only the issues higlighed by flake8 which is integrated in the GitHub Actions workflow was deemed as relevant. In an early commit of the code, there were many code quality issues as shown in Figure 6. Because some of these issues would be very tedious to manually correct (seeing as they were mostly whitespace), autopep8 was downloaded to format the code in VSCode. This was able to fix most of the code quality issues but there were still some that had to be fixed manually such as lines being too long. The final linter results is shown in Figure 5. There are still some code quality issues in the final submission such as my hei_update and entry_update functions being too long. However, due to the nature of a Flask app and a route decorator, it is not possible for me to refactor this function into smaller functions as that would not allow for correct route definition.

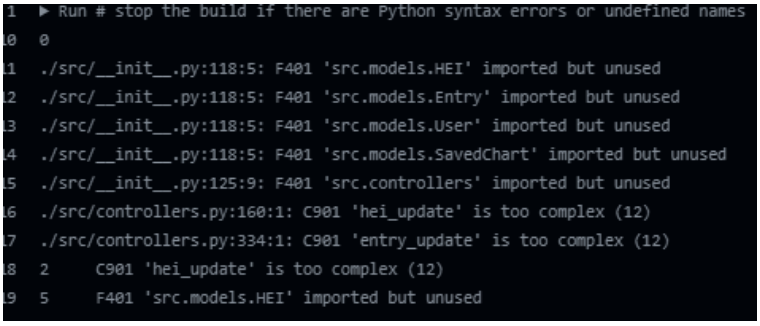

Figure 5 Code quality issues according to flake8 linter in GitHub actions workflow in final commit

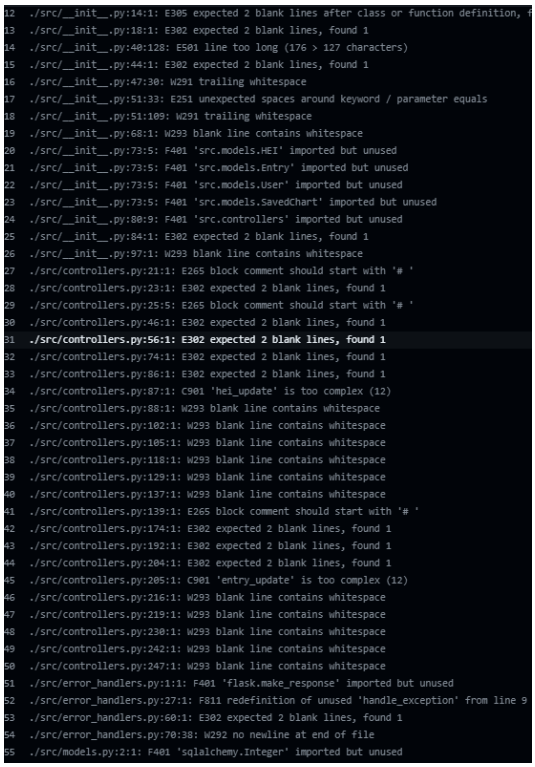

Figure 6 Code quality issues according to flake 8 linter in GitHub Actions workflow in early commit

References

Acknowledgement of the use of AI

In the development of the REST API, AI was used to generate, refactor and debug the code used for the application and the tests. Github Copilot v1.159.0 (GitHub, https://github.com/features/copilot) was downloaded as an extension in Visual Studio Code (the IDE I was using) and was able to automatically generate code based on the code already in my files and modules. Copilot was used to generate application code and test code as well as generate comments and doc strings for some functions. Copilot did not influence my code, the AI was influenced by the code I already had. ChatGPT-3.5 (Open AI, https://chat.openai.com/) was also used for debugging and refactoring of code. ChatGPT influenced the quality of my code as it was used to improve its adherence to DRY and good code design principles.

Dataset

The original dataset has the following details:

Title: Estates management by academic year and HE provider
Location: UK

Academic years: 2015/16 to 2021/22
Data source: HESA
Data file canonical link: https://www.hesa.ac.uk/data-and-analysis/estates/data.csv

The data used in the RESTAPI has been altered from the dataset above.