M4: Testing Update Report

Weather-or-not

https://github.com/SevNielsen/Weather-or-not

April 1st, 2024

Implemented Testing:

Phase 1: Unit Testing

The team has implemented the *pytest* framework for testing our flask application to modularize and test specific parts of the code. *Pytest* has allowed us to test individual parts of our code without altering the rest, along with setting up a trial database using SQLite to test the functionality without altering our existing database. SQLite is used for faster testing as it only creates a single-use database for the test that will be reset for each test case. In unit testing, we are testing all python code included in our tests, focusing on the database model.py and the utility files that perform functionality on data models. The system had to undergo heavy code refactoring and file organiztion to handle large-scale unit testing implementation from scratch. This process is far better when implemented from the beginning of production and when thought is put into how code will be tested when being built.

Phase 2: Integration Testing

For integration testing the team extended the “Flask-Testing” framework to test whether we are receiving the expected response when a new site is called, model manipulation or API request is made to a third party. These tests are located in the functional folder within our repository. These tests have been more difficult to implement as not all our code currently passes the unit tests. Thus it is difficult to continue moving forward with integration testing however much integration testing is mixed with manual testing and maneuvering of the web application. Our code is set with try-catch statements for manual testing purposes, this allows us to navigate the site and see the responses such as an HTTP call or a flash to help us further deduce the outcome. Further changes to our integration testing techniques will need to be made such as remaining more agile and updating code more frequently with tests included.

Phase 3: Functional Testing

Future steps for the team involve implementing playwright software to test the user experience and functionality of the web application. The software is used to simulate user interaction with the website to automate behaviour to test key components. This feature will allow us to test the end-to-end functionality of the web application to ensure there are no bugs or issues when published.

Phase 4: Load Testing

Once the app has been thoroughly tested for bugs, and logic errors, and refactored for optimized functionality we will begin load testing using the external software tool Locust. Load testing will ensure our web application reaches our non-functional efficiency requirement to withstand 10,000 users at once. Load testing will be the final step before publishing the application for public user and beginner user testing and feedback.

Phase 5: Continual Integration & Deployment

For continual integration, we have been working with GitHub Actions to protect the main branch, such as requiring another member to review code and designated tests are run on the code before it can be pushed to the main branch. These tests currently run quickly and ensure new additions attempting to be merged into the main branch will not break the application, or cause new bugs. Learning how to use GitHub Actions has been challenging and rewarding as it is a powerful tool that can be utilized to ensure our code infrastructure remains strong. Further use of automation will be implemented to shift tasks on the GitHub board to better manage the project. Currently, we are running a GitHub action .yml file to ensure our python code is syntax-free when uploading to our main file, the file runs flake8 to lint the python code.

Phase 6: Test Coverage & Refactoring

The team is continually conducting coverage analysis on the source code to fill in the gaps of testing and ensure the app functions without bugs. Where possible we are making efforts to refactor code to reduce coupling and increase cohesion, this is primarily being done through file organizing and management. Through file organization, it has been easier to identify opportunities to refactor code that adheres to the SOLID design principles.

Tests so Far:

Functional

test_auth.py

Within test_auth.py we are testing route functionality, user input, and database manipulation. For example, we are testing functionality to add a user to the database, through the signup feature, login, and logout. Along testing route functionality to test the security of our service ensuring users can only vizulae data when logged in. All these tests are currently passing. This testing class shows us how far along we are with working implementation as we have the test routes pre-defined and waiting for completion.

Unit

test_models.py

The test models class creates a test database that information is then manually added to in the code for testing. After we have filled out the database to how we want, we are asserting or expected outputs to ensure the data is being stored in the db correctly.

test_weather_utilities.py

This file holds the majority of functionality in the application such as making third-party calls to OpenWeathers API and conducting data analysis. These tests consist of setting up a mock API call and specifying what the JSON output will be for us to assert true if the data is being collected and manipulated properly. For example, to test our fetch coordinates from a city API call we are mocking the request URL, and providing a specific JSON response latitude and longitude for us to assert if true. Within this process, we can also test our code's ability to handle errors in response such as when we do not receive a 202 code to display an error message.

Example:

```python
def test_fetch_weather_data_success(mock_requests, mock_env):
    city = "Paris"
    weather_url = f'https://api.openweathermap.org/data/2.5/weather?q={city}&appid=test_api_key&units=metric'
    forecast_url = f'https://api.openweathermap.org/data/2.5/forecast?q={city}&appid=test_api_key&units=metric'
    mock_requests.get(weather_url, json={"cod": "200", "weather": [{"main": "Clear"}]})
    mock_requests.get(forecast_url, json={"cod": "200", "list": [{"weather": [{"main": "Clouds"}]}]})
    weather_response, forecast_response = fetch_weather_data(city)
    assert weather_response is not None
    assert forecast_response is not None

def test_fetch_weather_data_failure(mock_requests, mock_env, app_context):
    city = "UnknownCity"
    weather_url = f'https://api.openweathermap.org/data/2.5/weather?q={city}&appid=test_api_key&units=metric'
    forecast_url = f'https://api.openweathermap.org/data/2.5/forecast?q={city}&appid=test_api_key&units=metric'
    mock_requests.get(weather_url, json={"cod": "404"})
    mock_requests.get(forecast_url, json={"cod": "404"})
    weather_response, forecast_response = fetch_weather_data(city)
    assert weather_response is None
    assert forecast_response is None
```

Automation:

We implemented automation through GitHub Actions to increase security, automate testing, and manage files. Through GitHub actions, we have added branch protection by ensuring no one is directly allowed to commit to the main branch without having another member review the code before implementation, this feature can still be overpassed by admin when necessary. Further implementing Python builds stability, ensuring that all new code being implemented onto the main branch follows Python build and syntax requirements. We are now looking to implement automation to sort and filter issues, where issues are automatically assigned, sorted, and closed upon completion. We also can run our web application on multiple cloud services through automation such as AWS, Microsoft Azure, and Google Cloud to name a few, however, this will not be implemented until there is a purpose for cloud computing.

Ensuring Quality of Source Code:

Ensuring the quality of source code is a multifaceted process that requires both technical and organizational strategies. At the heart of these strategies is the principle of continuous improvement, which we have achieved through the following methods:

Code reviews are a critical component of maintaining high-quality code. They allow developers to critique and learn from each other, catching bugs and identifying potential improvements before the code is merged into the main codebase. This collaborative process not only improves code quality but also fosters team unity and knowledge sharing.

Automated testing, including unit tests, integration tests, and end-to-end tests, is essential for ensuring code reliability and functionality. These tests can be integrated into a continuous integration/continuous deployment (CI/CD) pipeline, allowing for automated testing of code changes in real time. This helps in quickly identifying and fixing errors, reducing the chance of bugs reaching production.

Adherence to coding standards and best practices is another pillar of high-quality code. Establishing a clear set of coding guidelines helps maintain code consistency and readability, making it easier for team members to understand and work with the codebase. This can be supported by using tools like linters and formatters to enforce these standards automatically.

By combining these approaches, our teams have significantly enhanced the quality of our source codes, leading to more reliable, maintainable, and efficient software solutions.

Summary & Review:

Project Summary:

Where are we now?

Currently, we are operating in week 12 of our sprint with a focus on the testing and refactoring stages of our code. Many new features are behind schedule due to a lack of frequent updates and testing iterations. Currently, our project has a complete signup, login, dashboard, and weather map feature with testing implemented. We are currently working on further data manipulation in graphs and charts to incorporate into our dashboard. The project is currently in the middle stages of development with many branches starting now looking for final touches and wrapping up of tasks and issues. The team has stopped adding new features and is now working to refine and refactor code for better readability, adaptability, and overall use.

Achievements:

- **Login/Signup/Recovery**

Big leaps in the user experience were made to the login/signup/recovery features providing users with a cleaner interface and stronger security linked to their account. Achieving multiple system requirements regarding security. Awaiting full implementation.

- **Dashboard**

The dashboard provided each member with a working hub to build new features. This was the first feature achieved and where we saw multiple pushes of separate features such as weather collection API calls and learning bootstrap in HTML, CSS, and JS to create our first function page.

- **Weather Map Layers**

The weather layers map has been seamlessly integrated into the source code from another online GitHub repository. By implementing pre-written code into our program that allows for the exact desired functionality and more has been a true learning example of the power of software

engineering. Learning how to manipulate preexisting code to our advantage has proven beneficial for the efficiency and functionality of our application. Completing the weather map with layers was a big achievement for the team.

Challenges:

Some challenges we faced as a team have been setting up our working environments by not fully utilizing VS code, and GitHub features. Communication across team members and self-delegation of tasks has been a challenge that is impacting the progress of the project. Another challenge faced has been the frequency of updates and competition of tasks on time.

Project Review:

Team Dynamics:

The team dynamic has been self-guided from the beginning with weekly scrum meets and sprint cycles of around 1-3 weeks. The group has been meetings every Tuesday, although often not all members have been able to attend although a weekly status scrum meeting report is produced for members to review if they missed information. Team members have been encouraged to work together more frequently to complete code and submit changes to GitHub as not submitting changes has been one of our biggest setbacks. Communication has been a struggle among team members as the channels are not often reciprocated when asked or communication is attempted remotely. Further in-person communication is required to attempt to mitigate this issue. Few conflicts have arisen among group members and team cohesion has strengthened despite lack of communication.

Scrum Process Evaluation:

The scrum process has been working well for the team as it has kept us on track and our goals aligned as team members. The weekly scrum document provides team members with all the necessary information required to move forward with that week's tasks and objectives. Meetings will begin to include KPIs to track the performance of individual team members to ensure we are remaining on track and submitting work. It is crucial that team members remain accountable for their code, so the rest of the team can continue to move forward.

Technology Stack Integration:

The project's development is running smoothly with the team's desired technological implementation. For communication, we are using Discord and Canvas, for for organization workload management we are using Google Drive, Github, and VS Code. File organization has been key to work delegation across group members and this is best done through GitHub and Google Drive. Github has been the most challenging for the team to learn however with the added tools in VS Code, working with Github has become much easier.

Future Outlook:

The team is focused on completing the project and delivering a refined product by the end of the school term. We will continue to focus on testing, and refining new features in the web application. With added emphasis on team cohesion and completing work as a group, as well as promoting more frequent updates and commits of code. Again team members need to be reliable, update, and test code frequently for the betterment of the application and overall team experience.