OPEN ENERGY DASHBOARD ENHANCED TESTING

A Capstone Report

Presented to the Faculty of CST 499 at

California State University, Monterey Bay

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Science

in

Computer Science

---

by

Mike Limpus

Fall 2022

EXECUTIVE SUMMARY

Open Energy Dashboard Enhanced Testing

by
Mike Limpus
Bachelor of Science in Computer Sciences
California State University, Monterey Bay, 2022

This project is an industry project brought to the college by Open Energy Dashboard, an application designed to help monitor resource usage in a web browser environment. The purpose of this capstone project is to assist the team with improving and expanding their unit testing suite. Some progress towards this goal has already been made by the OED team, so their work will be used as a basis for further development. Additionally, the team has requested assistance in repairing their failing automated continuous integration pipeline, a system which allows them to rapidly deploy changes to the software.

The eponymous application developed by Open Energy Dashboard is a free and open source web application for organizations to view information about their energy usage. It is designed to be free forever as a collaborative project that anybody can use. This project is a cost-effective and scalable platform to help organizations save money and reduce environmental footprint.

Table of Contents

PART 1 - INTRODUCTION

Introduction and Background

Project Name and Description

This report is for an industry project pitched by Open Energy Dashboard. Due to the fact that the application is developed and maintained solely by volunteer contributors, student involvement is common. Open Energy Dashboard has many projects set aside specifically for California State University, Monterey Bay Computer Science Capstone projects. Large applications such as Open Energy Dashboard develop unit tests, where groups of code are tested individually for functionality. To this end, the application team has requested assistance in augmenting their testing suite.

The Open Energy Dashboard team has three primary objectives in this project. In a recent update, the team added many new features. These features need to have corresponding unit tests written. Along with this, some existing tests need to be updated to be consistent with more recently implemented testing methodology.

The prominent issue that arises with free, open source projects such as these is finding consistent development. There are many reasons to contribute to open source projects, but the lack of a clear extrinsic motivator means that the pace of development is slow compared to commercial projects. Thus, developers may not spend as much time on projects such as these as they do on work to generate income.

The solution that Open Energy Dashboard has found is ingrained in the history of the project. OED originally started as a project at Beloit College, and as such much of its

contribution comes from students. These student contributions come from different schools, CSUMB being one of them. This capstone project will use what was learned about unit testing learned from CST 438 - Software Engineering and the work of other volunteer contributors as a guideline to add to an existing set of tests within Open Energy Dashboard.

Feasibility Discussion

Environmental Scan

There are many energy dashboards available today. However, unlike Open Energy Dashboard, these dashboards are usually proprietary and require a paid license. For example, MRI Software's eSight software is marketed as "Easy to use, engaging, and interactive dashboards to track and drive energy performance.". (Energy Dashboards, 2022). The software requires a license to use, as well as contact with a sales representative to see a demo and get a pricing quote. Much like Open Energy Dashboard, eSight offers customization to tailor the needs of an organization, as well as map visualization of the facility being monitored by the dashboard.

Justification

Users of Open Energy Dashboard rely on developers to implement new features and maintain existing code. In order for development to continue, code must be tested for reliability to meet a certain quality control standard, as outlined in the OED project principles. As a result of this capstone, users of Open Energy Dashboard and its developers will benefit from enhanced testing.

Ethical and Legal Considerations

Owing to the fact that OED is designed for monitoring energy and other resource usage, it is vital that the output to the end user is precise. To this end, it is vital that the tests being written are accurate. The consequences of a faulty product could mean increased energy costs or

elevated environmental impact. It is not likely that any underprivileged groups would be negatively impacted by this capstone. This project is intended to create energy conservation in a free and accessible environment, which serves to benefit everyone.

Under open source licensing, it is important to understand the licenses and their stipulations. Open Energy Dashboard is licensed under the Mozilla Public License. The MPL is a copyleft license that requires changes to code to be shared and distributed with a copy of the license (Mozilla, 2012). Any code written under this license must have a header that references this license. Code from some less restrictive licenses may be included in the project with that license's restrictions. Code from more restrictive licenses, such as the GNU Public License, may also be included. In this case, code must be distributed under both licenses (Mozilla, n.d.).

Long-term Enhancements

Under consideration that Open Energy Dashboard is an established, continuing open source project, the life-cycle of the product is nearly limitless. One specific example is in the testing for the project's client code. Currently, the front-end client for the project does not include any testing. The fact that this code was recently converted to TypeScript means that testing the code is less trivial than testing the server code. Early in this capstone project, an attempt was made to implement these tests, but it proved to be out of scope for the given timeframe. This could be returned to later, when time is less of a factor.

PART 2 - DESIGN AND TESTING

Design Requirements

Platform

Open Energy Dashboard is built in Node.js using JavaScript and TypeScript, using the React, Redux, and Express frameworks. The testing is implemented using the testing frameworks Mocha and Chai, which are used simultaneously to write tests. Per OED's recommendations, development was done using the Visual Studio Code text editor, using their Docker virtual container. The client recommends this platform as it ensures that all developers are effectively working in the same machine setup with the same software dependencies.

Major Functions

Over the course of this capstone, two major tasks were assigned by Open Energy Dashboard's project manager. First, I was asked to create a means for tests to be written for the new TypeScript code added to the last major update. Other volunteers are currently using this code to add new features and functionality, so it is imperative that this code be tested thoroughly. The client's specifications for this task are as follows:

- Tests should be easy and automated
- Ideally, tests should be written using Mocha and Chai
    - This is due to the fact that OED already uses these frameworks for testing
    - Changing packages would not be ideal as it would create some inconsistency in test development
- It would be preferable to use a TypeScript-to-JavaScript interpreter such as ts-node

OED uses many web APIs to send data to be plotted by its graphs and maps. The second major functionality I was asked to create was an update to one of their existing test files. The client requested that I rework the tests for one of the APIs to implement generalized testing data, rather than use a single value. Specifically, the client has requested:

- A jig which allows developers to
    - Load data into OED using a CSV file
    - Test the data being returned is as expected using another CSV file
- Tests which verify the functionality of both OED's APIs and the jig itself
    - These tests should include varied reading time ranges and rates

Usability and Evaluation Planning

Usability Test Plan

After consulting with the client, we decided to set a meeting to go over the completed work on a weekly basis, rather than present all completed work near the end of the term. This was to allow for regular feedback and advice. We met every Saturday for two hours, on average. We also communicated via instant messaging on a more daily basis, for more specific guidance in completing these tasks. There was no specific criteria involved, other than the OED developer guidelines. Rather, the client would review work and provide suggestions and guidance dependent on the work shown.

Usability Evaluation

At the end of the first week of working on the first given task, the problem had proven to be more significant than expected. Work on this problem continued into the end week two, at which point an agreement was reached with the client that focus should be shifted towards the second task for the time being, and a report on the findings of the previous two weeks was

written and added to the GitHub issues page. For the next three weeks, work progressed on the second given task, with the nature of the testing jig being updated based on client evaluation. At the end of week five, the final steps for the project were determined and implementation began based on this evaluation. It was decided that some final style and formatting changes would be made, and a formal pull request would be made on the main Open Energy Dashboard GitHub repository.

PART 3 - FINAL DISCUSSION

Timeline and Budget

Timeline

- Week 1
    - Met with the client for the first time. Discussed creating a testing jig for creating tests for TypeScript.
    - Began investigating methods to use Mocha in TypeScript. Created a small test project to ensure this would work in theory.
- Week 2
    - Continued research and experimentation with TypeScript jig. A significant amount of time was spent on this issue. Was able to narrow down a few possible reasons this would be happening.
    - Wrote a detailed report on the problem on the GitHub issues page.
- Week 3
    - Client agreed that TypeScript jig was out of scope for this project. Began work on jig for Readings API instead.
    - Updated the existing tests in the Readings API test file to be more comprehensive.
    - Wrote tests to check formatting of returned JSON from API.
    - Began early work on the test jig.
- Week 4
    - Client supplied CSV files to support large test data sets.
    - Jig completed, tests created using jig.

    - Wrote detailed instructions for other developers to follow as a comment in the source code.
- Week 5
    - Wrote up draft pull request on GitHub
    - Encountered major bug within the OED test suite which was causing tests to fail
        - Was able to correct this after extensive discussion with the client.
    - Finished pull request, ensured all CI/CD tasks completed

Timeline Discussion

In the capstone proposal, the milestones were defined as: Update existing unit tests, write new tests using varied data, investigate and fix GitHub Actions failing, and submitting to client for approval. The GitHub Actions milestone was ultimately resolved by another team working on their capstone for California State University, Monterey Bay. Updates to some existing tests, as well as new ones, were able to be completed. However, this fell behind schedule due to the aforementioned issues with TypeScript. In week 5, the work was able to be submitted to the client and approved for merging into Open Energy Dashboard.

Budget Discussion

Open Energy Dashboard strictly relies on free and open source software for development. As such, there were no associated costs involved with this capstone.

Final Implementation

The core part of this capstone project was the testing jig created for the Readings API. The Readings API is the part of the codebase which sends readings values from the database to the web client to draw all of the charts and graphs displayed by OED. It is important to test this,

as a failing Readings API would cause the program to display incorrect readings to the user, which could influence decisions in regard to energy usage that could end up costing money.

The first major addition was to update the existing tests for the API. The existing tests were only checking if one of the required parameters was present. For line readings, OED requires two parameters. These tests were updated to check for all required parameters. The test file was then restructured to run these types of tests after the new tests which were to be added, as they run relatively quickly. (see Figure A1)

The testing jig consists of two main parts: the JavaScript code and the required data sets. The JavaScript code is divided into multiple functions, allowing developers to only use parts of the jig if desired. The first of these functions allows users to insert test data into a mock database created by OED. The function is structured in such a way that the user can insert as many units of data needed for the test. (see Figure A2) There is a method which reads a CSV file of the expected values as a JavaScript object and parses out some metadata. (see Figure A3) Additionally, there is a function to compare an API reading to this expected values object, and throw an exception if they do not match. (see Figure A4) Finally, there is a helper function to create a date range using plain text. (see Figure A5)

To go along with the test jig, some actual tests were created in order to ensure proper functionality of the jig, while also being useful tests for the API itself. These tests checked readings of 50-100 values, created from an input of about 6000 values, with enough variance in the tests to ensure that the jig was functional.

Discussion

Issues Encountered

The first task assigned by the client was to implement Mocha testing in TypeScript. This was problematic as none of the standard methods for implementing Mocha into TypeScript projects were not working in OED. Some progress was made into this issue, but ultimately it was determined that this issue should be returned to.

Once work was mostly completed, it was discovered that when running the entire test suite another test file was causing the Readings API tests to fail. After some research, it was narrowed down to another test file modifying the Readings API source code without restoring it at the end of the test. This was a simple fix once discovered.

Reflection

A great deal was learned about contributing to large projects using version control technologies such as GitHub. Open Energy Dashboard follows a similar workflow as many open source projects using GitHub forks for development. The aforementioned issue also showed the importance of unit tests being atomic, meaning that they should not modify real data in the project.

References

Energy Dashboards. (2022, September 14). Retrieved from https://www.mrisoftware.com/products/esight-energy-management-software/energy-dashboards/

Mozilla. (2012, January 3). Mozilla Public License Version 2.0. Retrieved from https://www.mozilla.org/en-US/MPL/2.0/

Mozilla. (n.d.). MPL 2.0 FAQ. Retrieved from https://www.mozilla.org/en-US/MPL/2.0/FAQ/

Appendix A - Implementation Details

This appendix will include code samples and other materials related to the updated Readings API.

```
mocha.describe('rejection tests, test behavior with invalid api calls', () => {
        mocha.describe('for line charts', () => {
            mocha.describe('for meters', () => {
                // A request is required to have both timeInterval and graphicUnitId as parameters
                mocha.it('rejects requests without a timeInterval or graphicUnitId', async () => {
                    const res = await chai.request(app).get(`/api/unitReadings/line/meters/${METER_ID}`);
                    expect(res).to.have.status(HTTP_CODE.BAD_REQUEST);
                });
                mocha.it('reject if request does not have timeInterval', async () => {
                    const res = await chai.request(app).get(`/api/unitReadings/line/meters/${METER_ID}`)
                        .query({ graphicUnitId: 1 });
                    expect(res).to.have.status(HTTP_CODE.BAD_REQUEST);
                });
                mocha.it('reject if request does not have graphicUnitID', async () => {
                    const res = await chai.request(app).get(`/api/unitReadings/line/meters/${METER_ID}`)
                        .query({ timeInterval: ETERNITY.toString() });
                    expect(res).to.have.status(HTTP_CODE.BAD_REQUEST);
                });
            });
```

Figure A1: An example of the updated API rejection test.

```
async function prepareTest(unitData, conversionData, meterData, groupData = []) {
    const conn = testDB.getConnection();
    await insertUnits(unitData, conn);
    await insertConversions(conversionData, conn);
    await insertMeters(meterData, conn);
    await insertGroups(groupData, conn);
    await redoCik(conn);
    await refreshAllReadingViews();
}
```

Figure A2: The function to insert test data into the database

```
async function parseExpectedCsv(fileName) {
    let expectedCsv = await readCsv(fileName);
    expectedCsv.shift();
    return expectedCsv;
};
```

Figure A3: The function to load the expected values from a CSV file

```
function expectReadingToEqualExpected(res, expected) {
    expect(res).to.be.json;
    expect(res).to.have.status(HTTP_CODE.OK);
    // Did the response have the correct number of readings.
    expect(res.body).to.have.property(`${METER_ID}`).to.have.lengthOf(expected.length);
    // Loop over each reading
    for (let i = 0; i < expected.length; i++) {
        // Check that the reading's value is within the expected tolerance (DELTA).
        expect(res.body).to.have.property(`${METER_ID}`).to.have.property(`${i}`).to.have.property('reading').to.be.closeTo(Number(expected[i][0]), DELTA);
        // Reading has correct start/end date and time.
        expect(res.body).to.have.property(`${METER_ID}`).to.have.property(`${i}`).to.have.property('startTimestamp').to.equal(Date.parse(expected[i][1]));
        expect(res.body).to.have.property(`${METER_ID}`).to.have.property(`${i}`).to.have.property('endTimestamp').to.equal(Date.parse(expected[i][2]));
    }
}
```

Figure A4: The function to conduct the main portion of the test

```
function createTimeString(startDay, startTime, endDay, endTime) {
    const dateString = new TimeInterval(moment(startDay + ' ' + startTime), moment(endDay + ' ' + endTime));
    return dateString.toString();
}
```

Figure A5: The function to create a time range parameter