NAU Energy Dashboard:

Software Testing Plan

Version 1.0

Team: Save Watt

April 5, 2019

Sponsored by:

Jonathan Heitzinger & Dr. Truong Nghiem

Faculty Mentor:

Isaac Shaffer

Team members:

Madison Boman, Hyungi Choi,

Ian Dale, Brandon Thomas

This page was intentionally left blank.

Table of Contents

1. Introduction

1.1 Project Motivation

In the advent of alarming reports about climate change, there has been an ever-growing effort in favor of energy conservation. Climate change is a genuine concern for the future of our planet. Northern Arizona University (NAU) has always done its part to reduce waste, and promote a greener future. NAU is looking to take further action to improve sustainability by leveraging data collection and analysis.

For the last several years, Northern Arizona University has been collecting a trove of raw data related to the operation and energy consumption of its buildings. As of now, there is not a comprehensive way to analyze NAU's energy data. Current methods are cumbersome and require a great deal of technical expertise. There are four major issues with the existing process that our sponsors would like to solve:

- Manually retrieving data from sensors and the current database is time-consuming. Automatic retrieval of the desired data sets is preferred.
- Current tools do not allow for interactive graphs which show trends over customizable time frames.
- Broad statistics must be run using external tools. Some automatic statistical analysis is preferred.
- Exporting data to mutable file types is very difficult.

Our sponsors are looking to make the most of NAU's energy data. This project is sponsored by Jonathan Heitzinger and Dr. Truong Nghiem. Together they are looking to develop a web-based system which analyzes and elegantly presents NAU's building operation data. They hope to create a product that impacts campus-wide sustainability and facilitates the collection and analyzation of NAU's energy data. To accomplish this, we are developing the NAU Energy Dashboard.

1.2 Solution Overview

The NAU Energy Dashboard is a comprehensive web-based application which can retrieve, graphically display, export, and run basic analytics on NAU's collected data. We plan to simplify the existing process with a tool that abstracts away the complexities that stifle current research efforts. Our users would like the NAU Energy Dashboard to:

- Automatically handle building operation data retrieval.
- Graphically display data in a way that is interactive.
- Run applicable broad statistics on a wide range of data points.
- Cleanly export mutable CSV files for additional analysis.

This solution eliminates the need for sifting through endless data, presenting data using external tools, and using specialized tools for fundamental statistical analyses.

1.3 Software Testing

To ensure that the NAU Energy Dashboard is functioning properly, we will be performing a series of software tests. Software testing is the process of systematically analyzing the functionality of a software product. Testing will ensure that the actual functionality of the components that make up our system match the expected results. Since this is a public system, it is important that the system is defect free and can withstand use by many different users. We will conduct a series of tests which will fall under three categories:

- **Unit Testing**: Testing individual units/components of our system.
- **Integration Testing**: Testing combined units/components.
- **Usability Testing**: Testing the system with real users.

Each category of testing will allow us to test the system in a different way. The unit tests will ensure that the methods within each of the system's key classes are performing operations as expected. The methods we will test are those that deal with data. There are many components in our system that are provided by Django, so we will only test components that were created or highly modified by our team. Integration testing will assure us that data is being transferred between components as expected. Since our system is made up of python, javascript, and html we will need to test how all these components interact with each other. These interactions will mainly be in the form of variables passing from python to html to javascript via Django's render functionality. Finally, we will be greatly emphasizing usability testing. Since the NAU Energy Dashboard is a publicly accessible web application, it is important that the system is easy to use and fulfills the expected functionality. Our greatest improvements and insights to bugs will come from real user experience with the system.

In the following sections we will outline in detail our plans for unit testing, integration testing, and usability testing. In each section we will explain each category of testing, why it is important to our system, what tools we will be using, and how we will implement the tests. We will conclude with an overall timeline for testing our expected outcomes of testing as a whole.

2. Unit Testing

2.1 Introduction to Unit Testing

In this section, we will be discussing our plans for unit testing. Unit testing focuses on testing the individual components that make up the system. The purpose is to ensure that each unit of software performs as designed. Since our system is object-oriented, our units will be methods within the most important classes. To carry out the unit tests we will be using the unittest module built into the Python standard library. This library is the preferred way to write tests in Django. Using this library will also allow us to create an in-depth test coverage report. This report will tell us how much of our system is actually being tested, so we can gauge whether or not a critical amount of code is being tested.

We plan to focus on three major modules when conducting unit tests. These modules will be backend.py, clean.py, and conversions.py. Each of these modules provides key functionality when working with data. Since the data is our main concern, we seek to ensure that data is transferred and manipulated as expected. The modules that deal with data presentation will be covered in integration testing, as these involve multiple modules working together. Here in unit testing, we are looking to ensure the core of our system is working as designed.

In the following subsections we will outline each method being tested, equivalence partitions, boundary values (if applicable), chosen inputs, and assurances. Since many of these methods rely on instance variables, we will discuss equivalence partitions and inputs based on these instance variables rather than parameters if necessary.

2.2 Testing backend.py

This module holds two classes, StaticDataRetriever and BackendRetriever. These classes are responsible for gathering data from NAU’s energy database and our backend database respectively.

Class: StaticDataRetriever

Method	Equivalence Partitions	Chosen Inputs	Assurances
update_buildings(self)	‣ self.b_identifiers = invalid_type ‣ self.b_identifiers = invalid_form ‣ self.b_identifiers = valid dict of building info	‣ self.b_identifiers = 7 ‣ self.b_identifiers = { “B1”: [“Name”, “Alias”, “other”] ‣ self.b_identifiers = { “B1”: [“Name”, “Alias”], …}	‣ Returns error for invalid_type ‣ Returns error for invalid_form ‣ Creates rows in Building table for each building

Method	Equivalence Partitions	Chosen Inputs	Assurances

update_sensors(self)	‣ self.log_dict = invalid_type
	‣ self.log_dict = invalid_form
	‣ self.log_dict = valid dict of Sensor info
		‣ self.log_dict= 7
		‣ self.log_dict = { “Blue” : “Banana”, …}
		‣ self.log_dict = { 123_456: [“B_num”, “Description”, “Name”, “TrendID”, “Type”], …}
			‣ Returns error for invalid_type
			‣ Returns error for invalid_form
			‣ Creates rows in Sensor table for each sensor

make_log_dict(self)	‣ self.connection = valid_sql server ‣ self.connection = invalid_sql server	‣ self.connection = valid sql connection to server ‣ self.connection = 12	‣ Return a dictionary containing information about buildings ‣ Return error for connection not made
__create_trend_string(self, log_id)	‣ log_id = invalid trend log string ‣ log_id = invalid_type ‣ log_id = valid trend log string	‣ log_id = “123_4D6” ‣ log_id = 65 ‣ log_id = “123_456”	‣ Returns error for invalid log_id format ‣ Returns error for invalid log_id type ‣ Returns log string padded with zeros
get_log(self, log_id)	‣ log_id = invalid trend log string ‣ log_id = invalid_type ‣ log_id = valid trend log string	‣ log_id = “123_4D6” ‣ log_id = 65 ‣ log_id = “123_456”	‣ Returns error for invalid log_id “Not Found” ‣ Returns error for invalid log_id type ‣ Returns dictionary of log values from database

Class: BackendRetriever

Method	Equivalence Partitions	Chosen Inputs	Assurances
getBuildingStrings(self)	‣ Buildings = invalid_type ‣ Buildings = list of models.Building	‣ Buildings = 43 ‣ Buildings = [<Building 1>, ..]	‣ Returns error for invalid buildings ‣ Returns a list of strings representing the building models
get_data(self, building, sens_type, init_date, fin_date, incr=1)	‣ Building = invalid_type ‣ sens_type = invalid_type ‣ init_date = invalid_type ‣ fin_date = invalid_type ‣ Incr = invalid type ‣ All valid inputs	‣ Building = 74 ‣ sens_type = 32 ‣ init_date = 123 ‣ fin_date = 432 ‣ Incr = “Hello” ‣ Building = models.Building, sens_type = “Electric”, init_date = datetime.datetime, fin_date = datetime.datetime, Incr = 45	‣ Returns error for invalid building ‣ Returns error for invalid sensor_type ‣ Returns error for invalid date ‣ Returns error for invalid date ‣ Returns error for invalid Increment ‣ Returns tuple of dates and values for parameters
get_num_strings(self)	‣ Buildings = invalid_type ‣ Buildings = list of models.Building	‣ Buildings = 43 ‣ Buildings = [<Building 1>, ..]	‣ Returns error for invalid buildings ‣ Returns a list of strings representing the building numbers

2.3 Testing clean.py

This module holds the cleaner class. This class is responsible for parsing inputs from the user as they explore the website.

Class: Cleaner

Method	Equivalence Partitions	Boundary Values	Chosen Inputs	Assurances
get_datetime(time)	‣ Time = invalid_type ‣ Time = invalid_format ‣ Time = valid_format	‣ None	‣ Time = 12 ‣ Time = “12-1-17 4:30:45PM” ‣ Time = “February 2, 2017 5:37:21PM”	‣ Returns error for invalid_type ‣ Returns error for invalid format ‣ Returns time as a datetime.datetime object
get_build_info(str)	‣ Str = invalid_type ‣ Str = invalid_format ‣ Str = valid_format	‣ None	‣ Str = 12 ‣ Str = “Hello” ‣ Str = “Adel Mathematics (B26)”	‣ Returns error for invalid_type ‣ Returns error for invalid format ‣ Returns a tuple containing (“Adel Mathematics”, “B26”)
split_urls(builddata, flag)	‣ Buildata = invalid_type, Flag = valid_input ‣ Buildata = valid_input, Flag = invalid_type ‣ Buildata = invalid_format, Flag = valid_input ‣ Buildata = valid_input, Flag = invalid_form ‣ Buildata = valid_input, Flag = valid_input	‣ None	‣ Buildata = 12, Flag = valid_flag ‣ Buildata = valid_buildata, Flag = 12 ‣ Buildata = “Buid=“B60”, Flag = valid_flag ‣ Buildata = valid_buildata, Flag = “Sensor” ‣ Buildata = valid_buildata, Flag = valid_flag	‣ Return error for invalid buildata type ‣ Return error for invalid flag type ‣ Return error for invalid buildata format ‣ Return error for invalid flag format ‣ Return list which includes [Building Num, Utility/ Sensor, start time, end time]

2.4 Testing conversions.py

This module holds the conversions class. It holds many functions that are responsible for converting data between units. The functions here are of two types. They either solely take data as an input or they take data and some exchange rate. The exchange rate could be a price or other proportion necessary for conversion. Here we show how we will test functions that fit either type.

Class: Conversions

Method	Equivalence Partitions	Boundary Values	Chosen Inputs	Assurances
kbtu_to_btu(data)* *Many functions in conversions.py follow this structure. They can all be summed up by this test.	‣ Data < 0 ‣ Data > 0 ‣ Data = invalid_type	‣ Data = 0	‣ Data = -10 ‣ Data = 30 ‣ Data = “30”	‣ Returns error for negative values ‣ Returns valid output ‣ Returns error for invalid value
dollars_to_btu(data, price)* *Many functions in conversions.py follow this structure. They can all be summed up by this test.	‣ Data < 0, price < 0 ‣ Data > 0, Price > 0 ‣ Data = invalid_type, Price = invalid_type ‣ Data > 0, Price < 0	‣ Data = 0 ‣ Price = 0	‣ Data = -10, Price = -2 ‣ Data = 30, Price = 3 ‣ Data = “Test”, Price = “Test” ‣ Data = 1, Price = -4	‣ Returns error for negative values ‣ Returns valid output ‣ Returns error for invalid types ‣ Returns error for negative price value

2.5 Summary

These tests will ensure that the core backend functionality is performing as expected. All of the methods above are responsible for generating the data that is passed throughout our system. Therefore, each must return values as expected and alert of any errors. In the following section we will move to the modules that make use of this core functionality. This will be our integration testing phase.

3. Integration Testing

3.1 Introduction to Integration Testing

In this section, we will be discussing our plans for integration testing. Integration testing focuses on testing the components in groups. The purpose is to ensure that each module interacts with the others as expected. This level of testing can show us where modules are failing to communicate properly. Django uses two main modules that ensure the aspects of the backend and the front end communicate as expected. These modules are models.py and views.py. These modules handle the transfer of data and presentation of data respectively.

Django uses the pyVows package to ensure that all contracts of communication are honored. This package can ensure that the models are constructed correctly and that views.py is rendering pages correctly.

In the following subsections we will outline each method being tested, equivalence partitions, boundary values (if applicable), chosen inputs, and assurances. Since many of these methods rely on instance variables, we will discuss equivalence partitions and inputs based on these instance variables rather than parameters if necessary.

3.2 Testing models.py

Models.py defines each of the tables in our backend database and their relative columns. If any of these entries are formatted incorrectly, then our system will not run correctly. In models.py we have 3 main tables, Building, Sensor, and User. The Building table describes our buildings. Each building has a name, number(Unique), alias(Unique), and id(Unique). These inform our Sensor model which stores sensor data associated to a given building. Finally our User table holds usernames, passwords, emails, and permission levels in order to manage authentication.

Using pyVows we can run two tests which ensure data is being created and stored properly. PyVows includes a method that builds a mock model and then asserts that all the fields are filled correctly. This method interacts with our database as well as an HTTPContext to simulate real interaction with data.

PyVows also allows us to use the method “should_be_cruddable(self, model).” This method updates and the database with dummy data and flushes it. With this method and a test html page we can ensure that any GET and POST methods that involve our models will work properly with our rendered pages.

3.3 Testing views.py

Views.py defines the functionality of each of our application’s pages. Each of these functions takes in a request and any URL parameters and renders a context based off of a given template. This is where everything comes together. Views.py consists of 18 different functions. Each is a variation of one of our 7 core pages. Here we can use a combination of pyVows and Django.test.Client to ensure that the application knows how to respond when given different URLs.

PyVows allows us to create mock models and contexts to be fed to the system via requests. These requests are read by views.py and we can then run assurance tests on the responses. These responses will be come in the form of HTTP response codes which we can interpret. In most cases we will receive a 200 or 404 response. If these tests return what is expected, we can be sure that each of our views is performing as expected.

3.4 Summary

Ensuring that each model and each view is functioning as expected will tell us that our system abides by its own communication contract. This, however, is not enough to ensure a system that works as expected. Our system is highly user centric, and thus must perform as they expect. This leads us to usability testing, which we will outline in the following section.

4. Usability Testing

4.1 Introduction to Usability Testing

In this section, we will be discussing our plans for usability testing. Usability testing focuses on the user's experience with the system. The purpose is to ensure that system meets the users expectations and is accessible to our target user base. This level of testing can show expose logical bugs and user errors that the system must be equipped to handle. It can also give us qualitative information regarding our system as a whole.

We have two main user groups for our system: general users and administrative users. General users may be students, staff, and the general public which may be interested in how NAU consumes energy. Administrative users are researchers and administrative staff that are interested in deeply analyzing each aspect of NAU's consumption. We will be conducting tests with each of these groups to gain insight on improvements that can be maid from either perspective.

We will mainly be using think aloud testing to gauge the user's experience as they use the system. Using this method, the user will be expected to express thoughts about the overall aesthetics of the system as well as their thoughts about the difficulty carrying out tasks. This testing method may be used alongside timed tests to measure the speed and ease of use of the system. In the following subsections we will outline how each of our main pages will be tested by users to gain insight into our system.

4.2 Testing homepage

Homepage testing will consist of 3 main parts: Registration, Log-in, and exploring aggregate data. The user will be asked to think aloud as they perform each of these tasks. The user will be given no instruction besides what they are meant to accomplish. This will allow us to gain insight into how intuitive the system is. The first time these tasks are performed, we will time the user to benchmark the speed of each task as well as how long a user might take to learn how to accomplish the task. The homepage tasks are described as follows:

- Create an account to log-in
    - The user should head to the Sign-Up! Link from the homepage.
    - The user should enter their credentials
    - The user should submit their information and be redirected.

- Log in using a previously created account
    - The user should head to the Log-in link from the homepage
    - The user should enter their credentials
    - The user should submit their information and be redirected.

- View aggregate data
    - The user should toggle between different unit types

    - The user should describe the information they are viewing in context

4.3 Testing building page
Building page testing will consist of 3 main parts: Navigate to a building page, View aggregate building data, and View data for a utility/sensor of choice. This will allow the user to carry out the main functionality of any building page while thinking aloud. The first and third tasks will be timed to gauge efficiency.

- Navigate to a building page
    - The user should head to the buildings menu in the header
    - The user should search for a building either by searching or using autofill
    - The user should be redirected to the building of their choice

- View aggregate building data
    - The user should comment on the information immediately presented to them on the building page
    - The user is free to toggle any of the buttons on the page and express their thoughts

- View data for a utility/sensor of choice
    - The user should use the graphing tool to see trends on data of their choice
    - The user should enter a date range, increment, and utility/sensor in order to graph their data.

4.4 Testing export/comparison page
Export/comparison page testing will consist of 3 main parts: Navigate to the export/comparison page, View data for utilities/sensors of choice, and Export data to CSV. This will allow the user to carry out the main functionality of these pages while thinking aloud. All tasks will be timed to gauge efficiency.

- Navigate to export/comparison page
    - The user should head to the appropriate link in the header
    - The user should comment on the initial presentation of the page

- View data for utilities/sensors of choice
    - The user should use the graphing tool to see trends on data of their choice
    - The user should enter a date range, increment, buildings, and utilities/sensors in order to graph their data.
    - The user should comment on the presentation of data

- Export data to CSV
    - The user should use the export button to receive their data as a .CSV file
    - The user should comment on the presentation of the .CSV file

4.5 Testing admin page

Admin page testing will consist of 3 main parts: Navigate to the admin page, Edit permissions, and Perform a manual update. This will allow the user to carry out the main functionality of this page while thinking aloud. All tasks will be timed to gauge efficiency.

- Navigate to the admin page
    - The user should head to the appropriate link in the header
    - The user should input their admin credentials
    - The user should comment on the presentation of administrative functionality

- Edit permissions
    - The user should navigate to the user side of the admin page
    - The user should promote or demote a user of choice
    - The user should comment on the process

- Perform a manual update
    - The user should be able to refresh the backend database with a click of a button
    - The user should click the “Update system” button
    - This will be timed to see how long an update might take
    - The user should then navigate to the main site to see what data may have changed.

4.6 Summary

These user tests will allow us to continue to improve the system and ensure our users are satisfied. These user tests will take place at the end of our development cycle. Here we will have three days of interviews followed by three days of refactoring. This will happen in three cycles such that we test the system for a total of 18 days. At the end of the testing period we may perform final refactoring in order to ensure a stable release of the system. In the following section we will briefly review the topics covered in this document and conclude with our vision of a highly usable product.

5. Conclusion

Sustainability is a growing global concern. Any step we can take to manage our consumption of resources more efficiently is a step in the right direction. Our mission is to create an interactive energy dashboard for NAU's facility services. As we work with our sponsors, Jonathan Heitzinger and Truong Nghiem, to create the NAU Energy Dashboard, we are ensuring that NAU positively impacts the future of our planet and is a leader among the Universities of the world.

In this document we have covered our plans for Unit Testing, Integration Testing, and Usability Testing. These series of tests will ensure that our system is ready for a stable release at the end of the development period. By thoroughly testing the main components of our system and gathering user feedback we will be able to deliver a maximally error-free, functional, and highly usable software product. In the end this will lead to a successful NAU Energy Dashboard.