Transition Plan

LEAP Data Management Tool

Service Learning Practicum 2014

Document Version 1.0

Deployment Plan

Based on discussions with our mentor and customer, the system will be deployed to Bluehost. Bluehost was chosen because since is a PaaS, it will have relatively easy management and deployment, and out of the other viable PaaSes, it is the one that meets the customers cost restrictions. A database add-on will be used.

Our system will need some changes and additions in order for it to work on Bluehost:

1. In the .htaccess file
   ```
   <IfModule mod_passenger.c>
   Options -MultiViews
   PassengerResolveSymlinksInDocumentRoot on
   #Set this to whatever environment you'll be running in
   RailsEnv production
   RackBaseURI /
   SetEnv GEM_HOME /home1/examplec/ruby/gems
   </IfModule>
   ```
   From: <https://my.bluehost.com/cgi/help/rails>
2. Modify the ~/.bashrc file
   ```
   export HPATH=$HOME
   export GEM_HOME=$HPATH/ruby/gems
   export GEM_PATH=$GEM_HOME:/lib64/ruby/gems/1.9.3
   export GEM_CACHE=$GEM_HOME/cache
   export PATH=$PATH:$HPATH/ruby/gems/bin
   export PATH=$PATH:$HPATH/ruby/gems
   ```
   From: <https://my.bluehost.com/cgi/help/rails>
3. Create Ruby on Rails Application which forces Phusion Passenger to see application
   From: <https://box431.bluehost.com:2083/frontend/bluehost/ror/index.html>
4. "/leap" route has to be "" in application_controller.rb
5. Need to change database.yml to match new MySQL DB
6. During MySQL dump import, need USE [database_name] in SQL dump
7. Change `leap`@localhost to `leapvada`@localhost in the SQL dump
8. In the root of the project folder, bundle install will need to be run when deployed or updating the code base

The customer has been informed that Bluehost will cost $133.37 per year for the systems specific hosting needs. The timeline for the deployment is on track: the original system has been successfully deployed. At our next meeting, on Wednesday, February 5th, our client will be confirmed LEAP's opinion on deployment. On Wednesday, February 19th, all information needed for deployment was compiled. On Wednesday, March 5th, the acquisition was completed, and by Tuesday, March 24th, the system will be fully deployed.

http://www.bluehost.com/

Master Test Plan

1. Intro - This master test plan is an overview of the methodologies and approach to testing the Data Management Tool for LEAP Virginia.
    1.1. Scope
    The test application is written using a Model View Controller design pattern inherent in the Rails framework. In order to cover the greatest extent of code, we plan to test each of the facets of the MVC pattern separately.
        1.1.1. Models and Database Operations
        Tests in this area will be ensuring that the rails ActiveRecord associations are properly written and database access occurs consistently and correctly. This will show that data entered into database can be trusted. Additionally, classes will be tested through unit tests to ensure that all instance methods produce expected results on predetermined inputs.
        1.1.2. Views
        Views will be primarily evaluated through inspection. Acceptance and Security testing will be employed to check to ensure that the specifications are met for what is displayed to the screen. We will also be testing the javascript functionality of the webpage to ensure that the appropriate content is displayed for given environments and given database data.
        1.1.3. Controllers
        Broad unit tests will be used to test controller functionality such that the correct variables are rendered to the view based on the function calls. The tests will trace the routes of calls by a user and ensure that the system handles the correct workflow as specified by the requirements document.
        1.1.4. Usability
        Testing in this sector will primarily be evaluated by the customer in alpha and beta testing periods. The main focus here will be on acceptance testing, to ensure that the necessary functionality agreed upon at the beginning of the year. This includes the ability to navigate to data reports, do queries on the data, and export data in appropriate formats.
    1.2. References
        1.2.1. LEAP Requirements Document
        1.2.2. IEEE Standard for Software and System Test Documentation
    1.3. System Overview (what we are building)
    The system is a data management platform to allow the Local Energy Alliance Program (LEAP) manage data scattered across different databases and files. The system will allow the upload of housing and utility data that LEAP currently does energy audits for. The system will allow the capability to view energy data gaps for each property, to

allow LEAP to analyze the efficiency of each property in its database. LEAP will also have the ability to acquire pre-defined reports through this system.

1.4. Test Overview

1.4.1. Organization

System Requirements Testing for the LEAP Data Management system will continue to be completed throughout development, and should be completed and verified by the time the system is configured and deployed. Unit Testing, Installation Testing, Security Testing, and Continuous Integration Testing will be completed throughout the later phases of development, specifically phases 5 and 6, while quality assurance is being completed, and will be completed by the time the system is configured and deployed.

Everyone on the LEAP team will be completing the testing, the results of the tests will be recorded, and the team will review these results to resolve any issues raised by the testing tasks. The team as a whole will be responsible for approving test products and processes.

1.4.2. Master Test Schedule

At the end of Iteration 10, the Specific Test Plans will be completed, and the team will begin testing. The milestone at the end of iteration 11 is for the system to be fully developed. In the iterations following iteration 11, all testing and bug fixes will be completed. If a test fails, that test and all dependent tests will be repeated on the updated system until the tests pass. Testing will ensure the system will be fully functioning by April 25th.

1.4.3. Responsibilities: who writes the test

The tests, if possible, should be written by someone other than the person who created the feature. Having a different perspective on the task helps to facilitate a more complete testing strategy (the idea being that the person who created the feature already tested it with a specific test case in mind, so by having someone else test it you can get a broader perspective on the matter).

1.4.4. Tools, techniques, methods: how (language specific)

First, ‘fixtures’ or pieces of sample data should be created for testing purposes. These will allow us to more effectively ensure test coverage. Fixtures are, effectively, ActiveRecord objects, just like things that are stored in the database, except that they are defined in the /fixtures directory. Tests are run using ‘rake test’ from the command line.

1.4.4.1. How unit testing works

Unit testing occurs in files created in the /test/models directory. Files are created there for each model that is created. ‘Assert’-type unit testing methods can be written there. It is good practice to have a test for each validation and every method your model uses.

1.4.4.2. SimpleCov

We can use SimpleCov to verify code coverage; that is, make sure that our unit tests cover as much as possible. SimpleCov is a gem that inspects the tests being run and reports results at coverage/index.html. This will be used to verify that our unit tests are complete, or as complete as possible.

1.4.4.3. Jenkins

To facilitate our continuous integration testing, Jenkins (a stand alone product that interfaces with ruby via a gem) will be used in addition to SimpleCov. Jenkins facilitates running CI tests and reviewing results efficiently. Jenkins relies on having a robust set of tests already developed, and if this isn't the case, CI testing will not be useful.

2. Details of Test Plan

2.1. Test Processes : Intro to types of testing levels.

2.1.1. Continuous Integration Testing

We will be using Jenkins as part of our continuous integration testing. We will add unit tests to key parts of older code and aim to keep newer code pushed with unit tests to ensure that the system is working at all time. Since the system is constantly being live tested, these unit tests will help provide great feedback every time someone accesses and uses the website, especially since the majority of the website is looking through the data sets we have set up.

2.1.2. System/Requirements Testing

- Functionality testing of the requirements will be done mostly through letting the client test out the project in alpha/beta release, leading them through the features and how to use them. If the client believes that the system does everything they desired, we will consider the system functionally complete.
- Usability aesthetics are not very important to our client, as long as they can reasonably use our system easily. In terms of aesthetics, our client is fine with how the current layout is, so for the most part this phase of testing can be considered already complete. Since our client's main goal is to more rapidly interpret their data through the use of our system, we will mostly be testing to be sure our customer feels that the system is readily useable for their purposes.
- Failure in the system will be severe for our user, as they use this system to interpret data. Thus, testing of the system will ensure that no significant malformed data or data loss occurs. We will do this through vigorous use case testing that will ensure the data comes into the system and is viewed by the customer in an accurate way.
- Due to the small number of users of our system and non need for rapid feedback, performance time is not a big concern for our system. Relatively few

testing resources will be dedicated to this area.

2.1.3. Acceptance Testing

See functionality testing in 2.1.2.

2.1.4. Unit Testing (validation)

Unit testing will test every method in our Property, RecordLookup, Recording, Upload and User models and controllers. It will also test validation where appropriate. We will use Test::Unit, Rails standard testing library, to make tests for our methods. With Test::Unit, the most applicable assertions to our system are asserting equality, sending, matches, and null-cases. For example, to test create and destroy methods, we would use assert_nil or assert_not_nil.

2.1.5. Installation Testing

We will test installation by following our installation manual on a standard, unconfigured linux machine. This will test to see if Rails gems, directory permissions, and database setup are properly managed. Further details will be in the installation manual when written.

2.1.6. Security Testing

The client's database system is meant for internal use only so we will test authentication and authorization. We will test all pages for any unauthorized access. Our authorization is handled by giving accounts administrator privileges and only allowing admins to add other users. We will check if anyone but our clients can log in to use the system and test to see if anyone can make an admin account. Other basics such as password encryption and secure input/output handling will be tested for as well.

2.2. Test Reporting Policy

2.2.1. How to document tests, Redmine, etc.

Test reporting will be relatively simple. Mostly, the documentation will be done in the code, notifying what the test is for. We will also provide larger documentation on Redmine that will be useful for any future users. The documentation will outline what each test indicates.

Continuous Integration Test Plan

1. Continuous Integration Testing
    1.1. Jenkins
        1.1.1. Jenkins, a stand alone product that interfaces with ruby via a gem, will be used to facilitate continuous integration testing. Unit tests will be added to key parts of older code. Newly pushed code will be auto-tested with unit tests to ensure the system is working at all time of the push. Since the system is constantly being live tested, these unit tests will help provide great feedback every time someone accesses and uses the website, especially since the majority of the website is aimed at looking through the data sets we have set up.
        1.1.2. Jenkins facilitates running CI tests and reviewing results efficiently. Jenkins relies on having a robust set of tests already developed, and if this isn't the case, CI testing will not be useful.
    1.2. SimpleCov
        1.2.1. We will use SimpleCov to verify code coverage and make sure our unit tests cover as much as possible. SimpleCov is a gem that inspects the tests being run and reports results at coverage/index.html. This will be used to verify that our unit tests are complete, or as complete as possible.

Specific Test Plan

1. Intro: LEAP Specific Test Plan
    1.1. This specific test plan adds details to the Master Test Plan and is an overview of the methodologies and approach to testing the Data Management Tool for Local Energy Alliance Program (LEAP) Virginia.
    1.2. The system is a data management platform to allow LEAP to manage data scattered across different databases and files. The system will allow the upload of housing and utility data for residences LEAP has completed energy audits on. Users will be able to view energy data gaps for each property, which will allow LEAP to analyze the energy efficiency of each property in its database. LEAP will also have the ability to generate pre-defined reports and complete queries in this system.
2. Unit Tests
    2.1. Unit Tests will use:
        2.1.1. Unit testing will test every method in our Property, RecordLookup, Recording, Upload and User models and controllers. We will use Test::Unit, Rails standard testing library, to make tests for our methods. With Test::Unit, the most applicable assertions to our system are asserting equality, sending, matches, and null-cases. For example, to test create and destroy methods, we would use assert_nil or assert_not_nil.
        2.1.2. Unit testing will test validation where appropriate.
        2.1.3. Unit testing occurs in files created in the /test/models directory. Files are created there for each created model. 'Assert'-type unit testing methods can be written there. It is good practice to have a test for each validation and every method your model uses.
        2.1.4. 'Fixtures' or pieces of sample data should be created for testing purposes. These will allow us to more effectively ensure test coverage. Fixtures are, effectively, ActiveRecord objects, just like things that are stored in the database, except that they are defined in the /fixtures directory. Tests are run using 'rake test' from the command line. This will ensure the appropriate content is displayed for given environments and database data.
        2.1.5. Broad unit tests will be used to test controller functionality such that the correct variables are rendered to the view based on the function calls. The tests will trace the routes of calls by a user and ensure that the system handles the correct workflow as specified by the requirements document.
        2.1.6. Classes will be tested through unit tests to ensure that all instance methods produce expected results on predetermined inputs.
        2.1.7. Unit Tests will ensure rails ActiveRecord associations are properly written and database access occurs consistently and correctly.

2.2. Determining which Unit Tests to write

2.2.1. The unit tests will be written using a Model View Controller design pattern inherent in the Rails framework. In order to cover the greatest extent of code, we plan to write unit tests for each of the facets of the MVC pattern separately.

2.2.2. Unit Tests will be written to test all components of the Requirements Document.

2.3. How will write the Unit Tests

2.3.1. The tests should be written by someone other than the person who created the feature. Having a different perspective on the task helps to facilitate a more complete testing strategy. The person who created the feature already tested it with a specific test case in mind, so by having someone else write the unit test for it, a bug is more likely to be caught.

2.3.2. The writing of tests will be split between all LEAP group members as equally as possible.

2.3.3. The team will review the tests to ensure all essential functionality is tested. The team as a whole will be responsible for approving tests and processes.

2.3.4. The tests will be documented in the code, notifying what the test is for. We will also provide larger documentation on Redmine that will be useful for any future developers. The documentation will outline what each test completes.

3. Installation Tests

3.1. We will test installation by following our installation manual on a standard, unconfigured linux machine. This will test to see if Rails gems, directory permissions, and database setup are properly managed.

3.2. Installation test will also be performed on the deployment target on Bluehost's hosting service, to ensure installation of the system on a managed hosting provider.

3.3. Further details will be in the installation manual when written.

4. Requirements Tests

4.1. These tests will ensure the system meets the requirements agreed upon by LEAP and the capstone team, described in the Requirements Document approved by both parties on March 5, 2014.

4.2. User Permissions and Login

4.2.1. The client's database system is meant for internal use only so we will test authentication and authorization. We will test all pages for any unauthorized access. Our authorization is handled by allowing only existing users to add other users. We will ensure no one but the appropriate users can log in and use the system, and test to see if anyone unauthorized can create a new account. Other basics, such as password encryption and secure input/output handling, will be tested as well.

4.3. Simple Data Visualization

4.3.1. Color coded Gaps Reports will show utility data for properties. Missing data

should be colored red, normal data should be colored green, and outliers in the data should be colored yellow (example: any meter read of "0").

4.3.2. Meter Read date should be captured.

4.3.3. Property views should include the number of days in the reading period so the data can be better analyzed

4.4. Data Management

4.4.1. Ensure no significant malformed data or data loss occurs-- the data should come into the system and be viewed by the customer in an accurate way.

4.5. Data Reports

4.5.1. Includes the ability to export reports to CSV files.

4.5.2. Utility Requests Reports

4.5.2.1. Should include Name, Customer ID, and Date

4.5.3. Completed Job Data Sets Reports

4.5.4. PRISM ready reports

4.6. Queries and Searching

4.6.1. Auto-complete and partial match functionality for customer names and addresses should work in both search bars.

4.6.2. The user should be able to search by first name, last name, and property address.

4.6.3. The user should be able to filter (ascending / descending) by name and test out date.

4.6.4. The user should be able to search properties based on selected installed measures.

4.6.5. Filtering results should be downloadable in a PRISM ready format.

4.7. Data Upload

4.7.1. Utility data should be able to be uploaded into the system.

4.7.2. Address sanitation should occur when the data is uploaded.

4.7.3. The system should not allow duplicate data to be uploaded into the system.

4.8. Editing Data

4.8.1. The user should be able to update customer name and account number.

4.8.2. The user should be able to update housing data.

4.8.3. The user should be able to update utility data.

4.9. Housing Data

4.9.1. For each property, a list of installed measures should be available. This includes boolean fields for the 18 possible installed measures and a free text field associated with each of the boolean fields where the user can enter contractor data or other information.

5. References

5.1. LEAP Requirements Document