



Person Finder | Google

September 11, 2019

Hardik Rana - 16C0138

Harshal Shinde - 16C0223

Overview


Google Person Finder is an open-source web application that provides a registry and message board for survivors, family, and loved ones affected by a natural disaster to post and search for information about each other's status and whereabouts. It was created by volunteer Google engineers in response to the 2010 Haiti earthquake. Google Person Finder is typically embedded in a multilingual Crisis Response page on Google's site, which also contains various other disaster tools such as satellite photographs, shelter locations, road conditions, and power outage information. Google also set up a Picasa account to allow people to submit photos of the name lists posted in emergency shelters, to be manually transcribed and entered into Google Person Finder.


Google Person Finder is written in **Python** and hosted on **Google App Engine**. Its database and API are based on the People Finder Interchange Format (**PFIF**) developed in 2005 for the Katrina PeopleFinder Project.


Scope

Unit testing is to be performed for the following files in the source code of the web-application. [<https://github.com/google/personfinder/tree/master/app>]

1. **api.py** - File which contains basic API's for reading/writing small numbers of records. It contains function for converting data in xsl format to CSV, function for import or export records in CSV / Excel format etc. For testing we will create some entry based on person model and check whether the functions in this file are working properly or not.

- 
2. **photo.py**- File which handles retrieving of uploaded photos for display. It contains functions like `create_photo`(which creates a new Photo entity for the provided image of type images), `set_thumbnail`(which sets thumbnail data for a photo), `get_photo_url`(Returns the URL where this app is serving a hosted Photo object) etc. For testing we will check its function by creating dummy image.
 3. **full_text_search.py** - File which contains functions for search for particular entry of a person in a database. For testing we will create some dummy entry of person in the missing person's database and check whether the function is working or not.
 4. **const.py** - File which contains constants that aren't specific to a particular module or handler. It contains dictionary for mapping from language codes to endonyms for all available languages and mapping from language codes to English names for all available languages. For testing we will take inbuilt `pyfif` constants and check whether the constants we are getting from `const.py` is matching with that or not.
 5. **create.py** - File which contains functions like `validate_date`(parses a date in YYYY-MM-DD format), `create_person`(create entry for person in the database based on the data user provided) etc. For testing date function we will take inbuilt `datetime` library based date and compare it with the date we are getting from function.
 6. **detect_spam.py** - File which contains handler for spam note detection and store bad word list and provide utilities to evaluate the quality of notes. It contains functions for estimating the probability of text being spam. For testing we will take few text words and check whether its spam probability is matching with the value we are getting from the function or not.
 7. **importer.py** - File which Support for importing records in batches, with error detection. This module converts Python dictionaries into datastore entities. For testing we will check each function of the file by comparing it with data values.

- 
8. **indexing.py** - File which supports for approximate string prefix queries. For testing we will check each function of the file by comparing it with data values. For testing we will check the functions against some prefix queries.
 9. **jautils.py** - File which contains Utility functions specific for Japanese language. It contains functions like `should_normalize` (which checks if the string should be normalized by `jautils.normalize()` as opposed to `text_query.normalize()`), `normalize` (which Normalizes the string with a Japanese specific logic.). For testing we will check the functions for some value whether they are giving right results or not.
 10. **jp_mobile_carriers.py** - File which contains functions like `get_phone_number` (which Normalize the given string, which may be a phone number, and returns a normalized phone number if the string is a phone number, or None otherwise), `is_mobile_number` (which Tests the given string matches the pattern for the Japanese mobile phone number). For testing we will check the functions for some value whether they are giving right results or not.
 11. **main.py** - File which is The main request handler. All dynamic requests except for `remote_api` are handled by this handler, which dispatches to all other dynamic handlers. For testing we will check each of the function's in it by creating dummy data wherever needed.
 12. **model.py** - File which is a Person Finder data model, based on PFIF. For testing we will check each of the function's in it by creating dummy data wherever needed. For testing we will check some of its function by creating dummy data wherever needed.
 13. **pfif.py** - File which converts between PFIF XML documents (PFIF 1.1, 1.2, 1.3, or 1.4) and plain Python dictionaries that have PFIF 1.4 field names as keys (always 1.4) and Unicode strings as values. Some useful constants are also defined according to the PFIF specification. For testing we will check some of its function by creating dummy data wherever needed.

- 
14. **resources.py**- File which contain all functions related to resource. For testing we will check each of the function's in it by creating dummy data wherever needed.
 15. **script_variant.py**- File which contains functions like read_dictionary(which reads dictionary file), romanize_single_japanese_word(which romanize a single japanese word using a dictionary) etc. For testing we will check some of its function by creating dummy data wherever needed.
 16. **searcher.py**- File which contains a utility class for searching person records in repositories. For testing we will check some of its function by creating dummy data wherever needed.
 17. **send_mail.py**- File which contains functions/class like EmailSender(which is a simple handler for sending a mail). For testing we will check its functionality by creating dummy data wherever needed.
 18. **tasks.py**- File which contains functions for scheduling next task, checking whether the repo is in test mode or not, counting total tasks etc. For testing we will check some of its functions by creating dummy data wherever needed.
 19. **user_agents.py**- File which contains all the utility functions which are used in different other files. For testing we will check some of its functions by creating dummy data wherever needed.
 20. **text_query.py**- File which contains class related to text-query and normalization. For testing we will check some of its functions by creating dummy data wherever needed.

Apart from this unittest we will also perform some **system test** like test for import csv functionality, test for download feed functionality etc..

Out Of Scope

1. **Regression Testing:** This testing is not being performed because the product does not have continuous updates. The products main features are already developed and minor updates are done periodically. So this form of testing is not being done.
2. **Spam detection:** There is no way for this app to detect whether the data entered by user is valid (it may be malicious false data) or not.
3. Web-app security and performance

Assumptions

The assumptions being made are as follows. Users who are using this web-app enters valid data. The web-app's database and API are based on PFIF data-model,so it can aggregate data from various other sites in crisis situations. The web-app uses Python as its core development language, so the python release which the application is working on should be stable as well. As the web-site is hosted on google app engine it should have a stable performance. Users need stable internet connection to use this site and to enter missing person's data or to find missing person.

Schedules

1. Preparing for this test plan by 11 September 2019.
2. Generation of Test Cases for each type of testing by 14 October 2019.
3. Test execution by 4 November 2019.
4. Preparation of Final Report by 8 Novemver 2019.

Roles and Responsibilities

The team members are Hardik Rana and Harshal Shinde. Hardik Rana and Harshal Shinde shall be splitting the unit test case implementation and execution as mentioned in the scope. We shall be splitting the system test case implementation and execution as per our

need. The load testing and usability testing will be carried out by Hardik and Harshal respectively.

Deliverables

The testing process will deliver test cases which will cover the functional and non functional tests for the application. These test cases can be further expanded to test other modules and functionalities. So the test will deliver a comprehensive test report along with the test cases code which will be coded by us.

Environment

- A 32-bit or 64-bit operating system like Windows 10 or Linux based.
- Mouse or other pointing device.
- Python 2.7.x
- Latest App Engine SDK for Python
- lxml - **lxml** is a Pythonic, mature binding for the libxml2 and libxslt libraries.'
- Csstselect- It parses CSS3 Selectors and translate them to XPath 1.0 expressions. Such expressions can be used in lxml or another XPath engine to find the matching elements in an XML or HTML document.
- OAuth2client - This is a client library for accessing resources protected by OAuth 2.0.
- Mock - mock is a library for testing in Python. It allows you to replace parts of your system under test with mock objects and make assertions about how they have been used.
- Pylint - Pylint is a tool that checks for errors in Python code, tries to enforce a coding standard and looks for code smells.
- Yapf - A formatter for Python files
- Test data will be provided in the respective test case files.
- Tools as mentioned in the next section, required are Python environment, pytest and unittest libraries.
- Web browser- Chrome or Firefox
- Network Connection [To open the website]
- No security requirements

Tools

To perform this testing we need to have a **Python environment** [because the source code of web-app is written in python] and **Google app engine**.

Python is an interpreted, high-level, general-purpose programming language. Python's design philosophy emphasizes code readability with its notable use of significant whitespace. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects.

Python is dynamically typed and garbage-collected. It supports multiple programming paradigms, including procedural, object-oriented, and functional programming. For Internet-facing applications, many standard formats and protocols such as MIME and HTTP are supported. It includes modules for creating graphical user interfaces, connecting to relational databases, generating pseudorandom numbers, arithmetic with arbitrary precision decimals, manipulating regular expressions, and unit testing.

The task of getting Python code to run on a website is a complicated one, but there are a number of different web frameworks available for Python that automatically take care of the details. **Google App Engine** is a free alternative and simplest to use, which uses a web framework called **webapp2**.

Unit-Test

Unit Testing is the first level of software testing where the smallest testable parts of a software are tested. This is used to validate that each unit of the software performs as designed. unittest supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework. The **unittest** module provides classes that make it easy to support these qualities for a set of tests. The unittest test framework is python's xUnit style framework. White Box Testing method is used for Unit testing. unittest framework supports following oop concepts: test fixture, test case, test suite and test runner.

Pytest

Pytest is a testing framework which allows us to write test codes using python. You can write code to test anything like database , API, even UI if you want. But pytest is mainly being used in industry to write tests for APIs. It is a robust Python testing tool, which can be used for all types and levels of software testing. pytest can be used by development teams, QA teams, independent testing groups, individuals practicing TDD, and open source projects. Pytest is a software test framework, which means pytest is a command-line tool that automatically finds tests we've written, runs the tests, and reports the results. Not only is the pytest output less verbose and more to the point - we also benefit from useful assertion information. In fact, projects all over the Internet have switched from unittest or nose to pytest, including Mozilla and Dropbox. Because pytest offers powerful features such as 'assert' rewriting, a third-party plugin model, and a powerful yet simple fixture model that is unmatched in any other testing framework. There is a compatibility layer in place so we can run libtbx tests with pytest and vice versa.

Locust

Locust is an easy-to-use, distributed, user load testing tool. It is intended for load-testing web sites (or other systems) and figuring out how many concurrent users a system can handle. The idea is that during a test, a swarm of locusts will attack your website. The behavior of each locust (or test user if you will) is defined by you and the swarming process is monitored from a web UI in real-time. This will help you battle test and identify bottlenecks in your code before letting real users in.

Locust is completely event-based, and therefore it's possible to support thousands of concurrent users on a single machine. In contrast to many other event-based apps it doesn't use callbacks. Instead it uses light-weight processes, through gevent. Each locust swarming your site is actually running inside its own process (or greenlet, to be correct). This allows you to write very expressive scenarios in Python without complicating your code with callbacks.

Defect Management

All the defects will be directly reported to the repository owner that is Google Inc github maintainer. There is an issue page where the test issues can be reported to. The link to it are <https://github.com/google/personfinder/issues>. Issues can be used to keep track of bugs, enhancements, or other requests. Any GitHub user can create an issue in a public repository where issues have not been disabled. We can open a new issue from a specific line or lines of code in a file or pull request. We can add code snippet where issue is found. There are many ways of doing that for this project on number of lines of code where we find the issue. To select a single line of code, click the line number to highlight the line. To select a range of code, click the number of the first line in the range to highlight the line of code. Then, hover over the last line of the code range, press Shift, and click the line number to highlight the range. We will need to type a title for the issue and we can also assign tags in terms of milestones, labels or assignees in which we can tag the severity of the issue we shall detect during testing.

When we open an issue from code, the issue contains a snippet showing the line or range of code we chose. We can only open an issue in the same repository where the code is stored. We can open a new issue based on code from an existing pull request. Issues are a great way to keep track of tasks, enhancements, and bugs for your projects. They're kind of like email, except they can be shared and discussed with the rest of your team. Most software projects have a bug tracker of some kind. GitHub's tracker is called Issues, and has its own section in every repository.

Risk and Risk Management

Risks that we might face are listed below.

- Lack of any personnel resources when testing is to begin like lack of libraries required to run the testing.
- Lack of availability of required hardware, software, data or tools which may include lack of OS environment which we have planned to test on or lack of required hardware for the testing process.

- Changes to the original requirements or designs by the currently developing Google developers could cause a delay in our work and test plan.

Risk Management that could be done are listed below.

- Requirements definition will be complete by September 10, 2019, and if the requirements change after that date then those changes will not be taken into consideration.
- The test schedule and development schedule will move out an appropriate number of days. This might not occur, as most projects tend to have fixed delivery dates.
- The number of tests performed will be reduced to overcome any delay as the quantity of test cases taken is quite large.
- The scope of the plan may be changed to make the testing process flexible and at the same time take in the new changes.

Exit Criteria

When all the enumerated test cases are coded and testing is performed using them then we will match the output with the desired output. If the desired output matches with the test output then those test cases will be considered passed. The list of passed and failed test cases will be prepared. This will be thoroughly looked into and a detailed report will be prepared. All the failed test cases will be reported to the developers. After finishing all of these process testing process will stop and hence that will be the exit criteria.