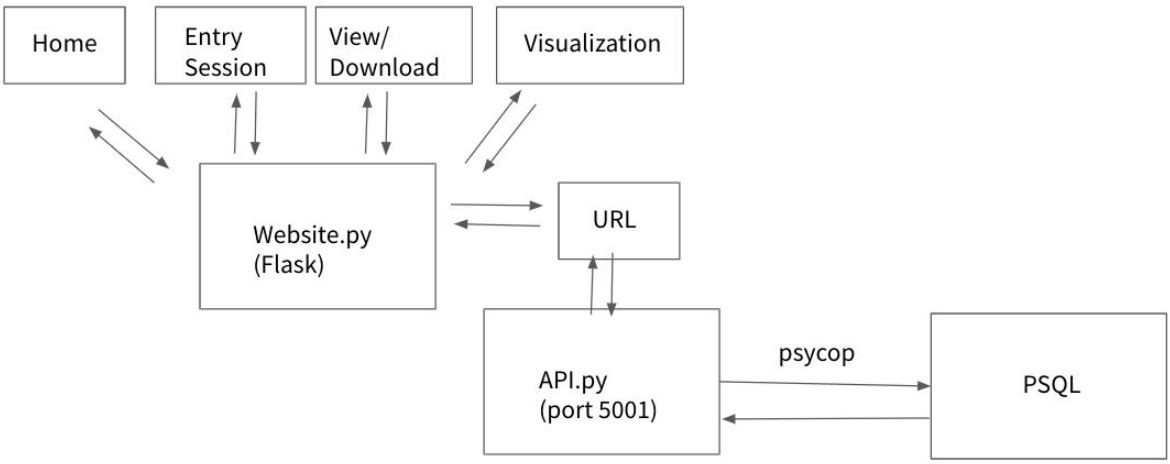
Developer Guide

Summary of Application Architecture

We've created a web application as a tool for data-inputting for the RFC group on campus. Our web framework is Flask and our database is Postgres. We have two different files to run our complete website. First is 'website.py'. The main purpose of 'website.py' is for mapping our html files to their respective URL's. Some functions also call our api to fetch data. The other python file we have is 'api.py'. 'api.py' holds the functions we use to make calls to our database. The file is organized so that there's one section for each html file. Private keys/information are stored in a configuration file that lives within the main computer responsible for hosting our website. As of right now, our Flask application and the database that stores the corresponding data are both hosted on a local computer(CMC307-08). We have a folder called static that holds all our CSS files, images and fonts used, and also some javascript files. For the most part, the javascript is embedded within the HTML file it interacts with. However, 'website.js' is its own file because it is used by all the HTML pages to check for log-in qualifications as well as control the navigation bar events. Our other folder is called templates, and that's where we store all of our HTML files. We have 6 pages total - 'data_entry.html', 'entry_session.html', 'home.html', 'login.html', 'view_download.html', 'visualization.html'. Specific information regarding these pages will be available in other parts of our documentation.



How to Run

To run our application, you must login to the RealFoodCalculator on CMC307-06 and redirect yourself to the RealFoodCalculator folder. Then you must start running 'api.py'. Once

you have the api started, as long as you have our application downloaded, you can host our website locally on your own computer. If downloading the complete application is too much of a hassle, you can also login to the RealFoodCalculator on CMC307-06 again and run 'website.py' as well.

'website.py' can be run anywhere, but 'api.py' must be run on the RealFoodCalculator account of the CMC307-06 computer for that's where the database lives. Once the computer gets cleared of the database, you must set it up again in order to have our website function correctly. Below is the 'How-To' for setting up another database.

To run application via remotely login to RealFoodCalculator:

- Open up Terminal
- ssh carletonAccount@mirage.mathcs.carleton.edu
- ssh RealFoodCalculator@cmc307-08.mathcs.carleton.edu
- cd RealFoodCalculator
- python3 api.py OR python3 website.py

To download application and run locally:

- Either ssh into the already set up database, or create your own to test with. Details below.
- Must have python3(https://www.python.org/downloads/), Flask(http://flask.pocoo.org/), simplejson(https://pypi.org/project/simplejson/), requests(https://pypi.org/project/requests/), and psycopg2(https://pypi.org/project/psycopg2/).
- Go to https://github.com/formidify/RealFoodCalculator
- Open up Terminal and find directory you would like repository to live in.
- Clone repository or manually download the zip file
- python3 website.py

How to Set Up Database

The database is a structured query language (SQL) database set up with PostgreSQL. Originally, this database was stored on CMC307-06 under a PostgreSQL database 'RealFood' and user 'RealFood'.

There is one table holding all individual receipt items as a row with each field the information required by the national RFC. The table is called "test_data_large" because we used purchased items from 2017-2018 with all costs set to $50.00 to preserve sensitive information.

In PostgreSQL, run this command to create the table:

```
CREATE TABLE test_data_large(
```

```
    month integer,
    year integer,
    description text,
    category text,
    product_code text,
    product_code_type text,
    label_brand text,
    vendor text,
    rating_version decimal,
    local boolean,
    local_description text,
    fair boolean,
    fair_description text,
    ecological boolean,
    ecological_description text,
    humane boolean,
    humane_description text,
    disqualifier boolean,
    disqualifier_description text,
    cost decimal,
    notes text,
    facility text
);
```

To place data from 'yourTestData.csv' into the database, run the following command: (NOTE: 'yourTestData.csv' should be formatted as the national RFC requires schools to submit data as of September 2018)

```
\copy test_data_large from 'yourTestData.csv' (format csv, null 'na');
```

Page-by-page information:

api.py: 'api.py**'** is the file we use to connect and interact with our database. We use flask to map URL's to specific Postgres commands. This file is predominantly used by the entry_session page, data_entry page, view_download page, and visualization page. The query calls are organized into sections for each page as you will see in the comments. The secret keys are stored in a private file saved on our server hardware.

website.py: 'website.py' uses flask to map the URL's for our front-end html files. Some functions also help fetch data from the api and manipulate it as needed on the relevant html file. This file also decides which website domain and port to share our application onto.

styles.css: This file styles the home, entry session, data entry, and view/download pages. The data entry page is styled using flex boxes to organize the information.

website.js: All html files are linked to 'website.js'. The top of the document handles how the website handles users being logged in. It first makes sure that if you're on a page within our web-app that's not the log-in page, your logged-in status, saved as a session storage, is true. If such is not true, you get redirected to the log-in page until you log in. If the logged-in status is set to true, the script also checks if the page you were redirected from was within one of our domains. It checks the credentials against a encrypted id and password with a key that's hidden in our database. The rest of the file handles simple navigation button events.

login.html: /rfc_login directs to login.html. This page takes in user input for the account ID and password, but all manipulations of the data for login validation is handled by 'website.js'. There's a separate css file for login.html called 'login_page.css'.

login_page.css: CSS styling for the login page

home.html: /rfc_home directs to home.html. This page holds the RFC introduction blurb and a nice picture of apricots.

entry_session.html: /entry_session directs to entry_session.html. This page provides link to enter data_entry window. When enter data is selected, data_entry.html will open in the same tab for optimal coordination between the two pages. Whenever the entry_session page is loaded, it accesses food data by parsing through localstorage string in JSON array format and populates html table with this data if the data exists. When edit is selected, table cells become editable by row. When save or delete is selected, the information is updated in the localstorage string. This string is what is submitted to the database when submit is selected. This string is cleared when cancel session is selected. Month, year, and entry session must be entered before submission, and data is sent to website.py. References javascript functions in script tag.

data_entry.html: /data_entry directs to data_entry.html. References javascript functions in script tag. This page has an html form "data-entry-form" that when submitted is received and handled by website.py. This form through website.py calls api endpoint /test_data_large to search for and display similar items that a user is searching for. Overall, this page has javascript code for error checking, checkbox usage, and submitting an entered item to entry_session.html by saving item as list of lists to localStorage "entrySession".

view_download.html: /rfc_view_download directs to the view_download.html. This html file sends api calls querying the searched result. It dynamically populates the HTML table, and whatever content is in that table is what gets exported when anyone presses the download CSV button. Aside from that, there's also a functionality to edit each row and transfer that change into the database. The edit button changes the element into ones in which the content is

editable, and the save button makes the api calls to the database, actually permanently storing the changes in our actual table.

visualization.html: /visualization directs to visualization.html. References javascript functions in script tag. Data is retrieved through flask calls to the database and dynamically visualized through the charts and some tables. Adding items/categories also send query calls to the database. All of the charts are implemented with the Chart.js library, and all chart attributes (such as color, line width, etc.) can be manually changed in the Javascript file. Current data shown in the charts and the tables are stored as JSON lists in the sessionStorage.

visualization.css: CSS styling for the visualization page. Slightly different than that of the other pages.

Troubleshooting:

- Restart Flask files (api.py and website.py)
- Check PostgreSQL database on CMC lab computer (if it is still on this computer), may need to be restarted if there has been a power outage or if the computer has been restarted.
- Try different web browser (Chrome preferred)

Next Steps/Areas for Improvement:

- More durable web server/flask upgrade (investigate Django, Apache, Waitress)
- More measures to improve security - another way of logging in.
- Consider web-based database (Google Firebase, Amazon Cloud Services)
    - Migrate off lab computer
- More error checking for data entry- consistent vendor names/product descriptions/spell-checking
- Add the ability to copy electronic receipts into the database
    - Perhaps through uploading a CSV file to website that inserts many items at once
- Expand project to other campuses