FoodMap

COS 333 Spring 2017

Product Guide

Overview

User Guide

Developer Guide

- Set-up for developers and contributors
  - Quick Start
- More detailed set-up and dependencies
  - Dependencies
- Email scraper (NodeJS)
  - Email fetching and parsing
  - Email processing
- Web app (Django)
  - Frontend (UI)
  - Backend (Database and API)
- Deployment in Heroku

Overview

This product guide is to help users and developers use our app. It is broken into sections by user and developer, and then within the developer guide by the role of the developer.

User Guide

Our product expands and improves upon the Free Food Listserv by reading and processing all events from it and presenting them on a map for easy visualization. We eliminate the need to subscribe to the Free Food Listserv and keep up with its emails. Our map shows users where they can go to get free food, and it provides a description of each event so that they can find food they want easily and quickly.

No installation or sign in is necessary; simply open https://foodmap333.herokuapp.com/ and all the information is freely available.

To see a summary of each event, consisting of the food offered there and how long ago it was posted, hover over a pin. To see a detailed description of the event, click on the pin.

For those with food to share, we provide a form for adding an event to our map. This quickly increases the event's visibility to a dedicated audience of hungry students with minimal effort -- much less than writing an email. Click the "Submit Food" button in the navigation bar to open the form. There you can select your event's location on campus from the dropdown menu and write a description. Submitting your event with just this information will, by default, show it on the map immediately, but you can also schedule your event

to occur later by specifying a later date and time. We also support scheduling events to repeat daily, weekly or monthly; optionally select one of these choices from the “Repeat” dropdown menu.

Developer Guide

The app is built using two main components: the Free Food Listserv email scraper (written in NodeJS) and the Django web app. These are documented for developer use in sections below: **Email Scraper** and **Web App**, respectively.

Before we get into the details of each component, we provide a “quick start” for developers. This will get your development environment set up so you can start work. Also, this can serve as a future reference if you ever mess up really badly (you know everyone does at some point) and need to set up your environment all over again from scratch. See the following **Set-up for developers and contributors** section. Also, *make sure to read the **Git Rules** section* for the rules and conventions of writing code for this project.

Set-up for developers and contributors

We have automated the set-up required for anyone who wishes to work on the project in a script `setup`. This automates installing dependencies, defaulting your environment to "development" mode for both Django and Node (we’ll explain what this means later in the section), and configuring the database.

Quick Start

If you just need to get the environment set up from scratch for the first time, run `setup` for your OS. For Mac:

```
./setup mac all
```

or for Ubuntu:

```
./setup ubuntu all
```

Even if you're only working on one end of the project (Node or Python), because they are so closely intertwined, you should have the Python virtual environment enabled whenever you're working on either end. Activate it with:

```
source venv/bin/activate
```

That's it! You should be all set to work on the project. You can verify that everything is set up properly by running the Django and Node JS tests:

```
python manage.py test
npm test
```

Note that you will need to activate the virtual environment again each time you work on the project.

More detailed set-up and dependencies

We recommend you do this quick start if you're getting the project set up for the first time. However, you can also choose to set up only the components you know you'll be working on.
`setup` takes two arguments: OS and mode. You can use the mode to specify what part of the set-up you want to do, as documented below (we use `mac` as the OS in all of these, for the sake of example):

- `./setup mac all`: Does the full set up (as in **Quick Start**)
- `./setup mac python`: Installs only Python packages
- `./setup mac node`: Installs only Node JS packages

- `./setup mac db`: Only initializes/configures/updates the database

Dependencies

The following dependencies are required (note that these are installed automatically by `setup`).

Django side (for development):
- Django 1.10.6: web framework
- Pillow 4.0.0: required for the database to be able to store images
- Selenium 3.3.1: required for browser automation to obtain latitude/longitude coordinates of locations

Node side (for development):
- google-auth-library 0.10.0: required for getting authentication to use google APIs
- googleapis 18.0.0: required for using google APIs
- sqlite3 3.1.8: required for interacting with SQLite3 database in development mode
- mocha 3.2.0: required for running unit tests

The app requires different configurations depending on whether it's running in the deployed, production version, or whether it's running in a local, development version. The above dependencies listed are required for the *development* version, for your convenience when you want to work on the project. Below we list the additional (or otherwise differing) dependencies used in the *production* version.

Django side (for production):
- Whitenoise 3.3.0: used to serve static files in Heroku
- Gunicorn 19.7.1: used to allow running multiple Python processes concurrently in Heroku
- dj-database-url 0.4.2: used for extracting database configurations in Heroku
- psycopg2 2.5.3: required for interacting with PostgreSQL database

Node side (for production):
- pg 6.1.5: required for interacting with PostgreSQL database in production mode

To determine whether to use the "development" or "production" configuration, we set two environment variables: one to indicate to Django which configuration to use, and one to indicate to Node. Django uses the `DJANGO_SETTINGS_MODULE` variable, which is set to `foodmap_proj.settings.development` in development mode or `foodmap_proj.settings.production` in production mode. This is built into Django, so it knows how to interpret this variable to use the correct configuration. See the documentation for this environment variable for details.

Node (to our knowledge) does not have a corresponding built-in variable, so we created one called `PROJECT_MODE` to serve the same purpose. It is set to `development` in development mode or `production` in production mode. We have programmed the scraper to interpret this variable to use the correct configuration in each mode.

Finally, we also list other libraries and other resources we used that do *not* require installation (they're mostly frontend CSS/JS libraries):
- Bootstrap 3.3.7: CSS framework for web pages
- jQuery 2.2.3: standard JS library for dynamic content (AJAX, animations, etc.)
- Leaflet 1.0.3: map framework/library for CSS/JS
- Leaflet Locate Control https://github.com/domoritz/leaflet-locatecontrol: Leaflet CSS/JSplugin for finding a user's geolocation
- Maps Icons Collection https://mapicons.mapsmarker.com: provides icons for map markers
- PhantomJS: virtual browser for web scraping (used to scrape data in `locations.json`)

- jQuery UI and jQuery Timepicker Addon 1.6.3: Provides a widget for selecting date and time in a form.

Email scraper (NodeJS)

To run the scraper, run the following command:

```
npm start
```

Email fetching and parsing

The scraper uses the Gmail API to access unread emails. The code in `oAuth.js` used for the authorization of the API is from the Node Quickstart Tutorial. The `client_secret.json` file must be in the `scraper` folder to get credentials for the API. The credentials for foodmap333@gmail.com are in the tar file, but they can also be retrieved manually by following Step 1 of the Node Quickstart Tutorial.

Each email from the Google API is formatted in the MIME standard. A good resource for understanding the format is Google's API Reference on Messages.

Email processing

Our algorithm works by checking whether each phrase, starting at a space, matches a word in a list storing numerous types of food. Once each food-phrase is identified, we also check whether any set of words separated by commas and/or the word "and" contains at least one food item; if it does, we can safely assume the other items are also food items.

If the phrase "all gone" is found in the email body, the scraper labels the email as a deletion request, and deletes the appropriate entry from the database. Otherwise, the scraper adds the entry to the database.

For location, we map from aliases to official building names. This allows the scraper to detect specific rooms that are often mentioned such as the LGBT center or Studio Lab and map them to Frist Campus Center and Fine Hall. The location is found via a largest matching of a substring search of all aliases.

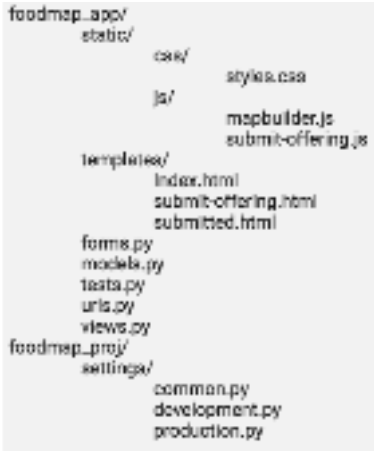
You can test the scraper by running unit tests written with Mocha, a Javascript test framework. The files are separated by the functions that they test.

Web app (Django)

This section describes the Django project backing the app. This is the part containing all the views, models, controllers, and other components of the app, except the Free Food Listserv scraper (that is kept separate -- see the previous section **Email Scraper** for details).

This section is divided into the frontend (UI) and the backend (database and models, web API into the database, and controllers). Before we get to these, we provide a brief overview of the directory structure:

`foodmap_app/` contains the components of the app: the models, views, and controllers. This is where both frontend and backend developers will spend most of their time. `foodmap_proj/` contains the configuration settings that apply to the entire project. Generally, these configurations do not need to be changed, but by default, Django keeps all settings in one file `settings.py` directly within the `foodmap_proj/` directory. To divide development and production settings, we break this into multiple files, and store them all in a `settings/` subdirectory. `common.py` contains settings common to both development and production modes, whereas `development.py` and `production.py` contain settings specific to



development and production modes, respectively. Each of these settings is documented in Django settings documentation.

Frontend developers will deal with the files in `templates/` and `static/`. These contain HTML, CSS, and JS files that form the frontend. They should also use `tests.py` to write tests for UI elements in accordance with Django's testing documentation.

Backend developers will deal with the "loose" files in`foodmap_app/`: `forms.py`, `models.py`, `urls.py`, `views.py`. They will also write tests, as frontend developers would, in `tests.py`.

Frontend (UI)

static/ contains the javascript and css files, while templates/ contains the actual html files that make up our web app. We primarily use bootstrap for styling. To generate a more aesthetically pleasing and interactive map interface, we use the leaflet library. Explanations of the functions used from the leaflet js and css files are located in the Leaflet documentation. The code for generating the Leaflet map is contained in mapbuilder.js, which is loaded into the index.html for the website.

In order to populate the map, we make requests to the Django database for a particular entry, which contains not only the information to populate the sidebar and info window but also the geographic coordinates with which to plot the food item on the map. Further details on the Django database are located in the **Backend (Database and API)** section.

Backend (Database and API)

Database

The database is defined in `models.py`

For convenience, we created a script `setup_database.py`, which does initial configuration of the database and populates it with all locations on campus. Run this with:

```
python setup_database.py
```

Note that `setup` runs `setup_database.py` to do database configuration, so you never really need to run this directly. It is only documented here for completeness.

The database model is defined in models.py . We use a simple relational model, implemented in SQLite in the development version and PostgreSQL in the production version. It consists of three tables: two currently in use, and one defined for future features but otherwise unused.

The Location model defines a table to store all buildings on Princeton's campus, the post's id, and their geolocations.

The Offering model defines another table to store all food offerings/events. Each offering has a location, referenced from the Location model; the type of food, the time the food was made available; the full text of the email; and the thread id of the email. The image attribute is intended for future implementation of allowing users to enter images.

The Location and Offering models are the ones currently in use. We also have defined an OfferingTag model, which is not currently in use but has the role of assigning a categorical tag to an Offering. We planned to use this to send specific notifications based on user preferences.

Also, to prevent the database from overflowing, the `delete_old_offerings.py` is run daily and deletes entries that are no longer relevant.

API

We implemented a web API that allowed the frontend UI to pull data from our backend database. The UI only ever shows all food entries within the last two hours, or submits a simple food entry. So we have only two API calls; one for each of the operations.

GET /offerings/

Returns all food offerings less than (or exactly) two hours old, formatted as a JSON array in the following format:

```
[
  { "location": {
      "name": "Frist Campus Center", "lat": "41.0000", "lng": "76.000"
    },
    "offerings": [
      { "title": "Pizza, Cookies",
        "description": "Pizza and cookies at Frist right now",
        "minutes": 40 },
        { … },
    ]
  }
]
```

Note that the offerings are grouped by location - there may be multiple offerings at each location - but are not necessarily sorted in the array in any particular order. In particular, they are not necessarily ordered by their "minutes" parameter.

POST /submit-offering/

Provided with a submission from our form located at /submit-offering/ (format standardized by Django - see Django Form Documentation), inserts a food offering into the Offering table of our database.

These API calls are implemented as Django views, methods located in `views.py`.

When the page is loaded, it sends a query to the given url and receives in response a JSON file in the following format:

```
[{
  "location": {
     "name": "Frist Campus Center",
     "lat": "12.3456789",
     "lng": "12.3456789"
    },
  "offerings": [
{ "title": "Pizza",
  "description": "Come eat pizza at Frist!",
  "minutes": 40
},
{ … }]
}]
```

After parsing this JSON response, the information encoded in it is formatted with HTML and populated into the markers (pins). We do this by attaching the formatted content to the "popupContent" attribute of each marker. Then, the markers are added to the map.

Deployment in Heroku

All of the code is deployed in Heroku. The web dyno of Heroku is configured via Procfile to use Django. The Heroku Scheduler addon makes the scraper run every 10 minutes and makes the `delete_old_offerings.py` run daily. The environment variables are set up so that the code can detect that it is in production mode and can retrieve credentials from environment variables.

Appendix: Git Rules

Branches and Commits

There should be no direct commits in the master branch, only merges from the development branch. There should also be no direct commits in the development branch except to update this document. All other branches should stem from the development branch and should be merged to it.

Branch Naming Conventions

The name of each branch should be a 1 to 3 word summary of the feature separated by hyphen -. For example, the branch for the scraper should be named scraper.

Code conventions

General

- Indentation: Indent with spaces, indent size of 4.
- Line endings: Unix-style \n.

Python-specific

- Strings: Use single quotes.
- Header comments: Standard Python style. Immediately inside the function/class, enclosed in triple quotes, with newlines between triple quotes.