FoodMap

COS 333 Spring 2017

Final Report

The Evolution of FoodMap

FoodMap underwent several evolutions before arriving in its final form. At its inception, it was going to be a shortest path app to find the nearest food entry in your area, but after a few critical meetings, we quickly realized this was not feasible, as there was rarely more than one free food event at a time. Fortunately, we quickly formed a strong team dynamic and developed a new, more practical idea: a map of all the free food on campus. Our subsequent journey includes pleasant surprises and unfortunate complications with a healthy dose of user feedback, bringing FoodMap to its present form.

Design and Development Decisions

From the beginning, we knew that a primary use case for the app was for people who were on the go and wanted to see if there was any food near them as they were moving around campus. Obviously this would make pulling out your laptop cumbersome, so we recognized the necessity for our app to work well on a mobile phone. We considered using React Native to build FoodMap as a mobile app, but we decided the learning curve was too high, and did not want to cut out an entire demographic by only building an iOS or Android app. We instead chose to build a web app to be accessible to everyone, but because of its origin as a mobile app we always made sure to check at every stage that our app worked just as well on a mobile browser as it did on a desktop, and added the feature of location finding with this use case in mind.

We considered whether a user should be required to sign in with CAS authentication to use the map, but we ultimately decided against it; any email can be used to subscribe to the freefood listserv, not just Princeton affiliated ones, so we decided it was not necessary for us to implement, and we would rather make use easier.

The careful design of the database schema allowed faster connection of scraper to the database. Before we started the project, we wrote down the names of the tables, the names of the columns, and the types of the columns. This enabled us to work remotely without constant adjustments.

Automating setup saved us a lot of time and effort, especially since we were on different platforms (Linux, OS X, and Windows). Rather than having to manually determine how to install each of our numerous dependencies for each platform, we wrote one automatic setup script that takes your os as an argument. This was especially useful for instances when we accidentally corrupted directories and needed to restart from scratch. The virtual environment allowed us to contain all of our installations instead of interfering with our local environments where we each were working on our own projects. Automating setup was probably one of our most useful investments because of how simple it made standardization and development.

For version control, we used GitHub. This made it easier to prevent poor code from causing impossible-to-find bugs, but it did cause difficulties in that we had numerous branches that sometimes could not be automatically merged into each other. Additionally, it became confusing to merge it all into one product for deployment because we had so many small branches for working on specific parts of the app.

We prioritized building test cases before new features and kept test cases even if they had worked once, which turned out to be a valuable decision when we often found that code we thought was ready to deploy was failing one of our test cases, allowing us to prevent bugs from stacking.

One of the most significant parts of our project is our database, so we tested it extensively. We have tests that ensure we are retrieving the most recent offerings, that deleting is working correctly and that our table of offerings is being manipulated correctly. We also make sure database of locations is accurate and immutable.

When we started developing the scraper, it became clear that the scraper would require a lot of unit testing. It involved many edge cases since we could not control the input format, and having multiple tests allowed us to get food information from even the most oddly-worded emails.

As we developed new features, we made sure to update our tests to include them, including adding tests for our integrated form. We also introduced bugs intentionally to make sure we had tests to catch them.

Planning

We had several successful meetings early in the planning phase in which we determined what each team member's specialties were and how they could best contribute to the project. We then divided up work and determined how to transfer information between our input sources, the database, and the UI.

Our initial design took input solely from the freefood listserv, storing entries in a SQLite database and using Django to load entries into a Google Maps interface. However, this plan changed as we discovered more useful resources for the UI and backend. In the UI, we discovered the Leaflet API after observing the much more aesthetically appealing interface of the Rooms Tigerapp. On the backend, we were originally planning to use SQLite for the database

engine. The engine worked very well in local servers, but in Heroku, the database was malfunctioning. The problem was Heroku's ephemeral filesystem; SQLite runs on memory, so the data was not updated properly. For this reason, we had to change the engine to PostgreSQL in production mode. This triggered the division of project modes.

After deciding on the structure of the project, we created a timeline and divided up work evenly and early based on prior knowledge. Michael had prior experience with SQL, so he was given a backend role. Rachana had prior experience with UI work from HackPrinceton projects, so she worked on the front end. Nathan worked in more of a fullstack role, helping out with both the scraper and the UI. Ryan had previously used Node.js with the Google API to fetch emails, so he was assigned to the scraper. Ryan also had deployed web apps in Heroku in React.js, and since we thought it was a simple way to deploy our app, we decided to use Heroku. As a team, we wanted to make use of this project to learn a new framework, so we decided to use Django. We additionally needed a good framework to interact with the database, and Django seemed to be a good solution based on what we had heard in lecture and after our own research.

We hit our first difficulty when we realized dividing work up evenly is not always the best plan; many parts are dependent on previous elements. For example, within the UI, the person assigned to the info windows and sidebar could not implement anything until the map itself was implemented. This cut down on team efficiency, so when we reached the additional features portion of our timeline we decided to break up work generally into the scraper, database, and UI, with people assigned to each section rather than a specific task. Additionally, the design changes we made for aesthetics and compatibility forced us to discard many hours of work that had been put into developing the earlier frameworks.

Generally, our habit of meeting frequently and staying in communication when working remotely via Slack helped us to stay on track with our timeline and thus release our product early.

Beta Release

Given that we completed our minimal viable product early, we decided to release it and our *planned* additional features as our beta release, rather than trying to add on all the features we had in mind and then releasing, since we wanted to get feedback from actual users. This ended up being a valuable decision, since we were able to find out what the users wanted and implement those features, making our app more popular and useful for the campus and saving us the time of implementing less useful features.

One key feature we had planned to implement was filters, allowing you to only display specific types of food on the map. However, thanks to user feedback that there was never enough food entries on the map for filtering of map entries to be useful, we realized that our time would be better put towards a commonly requested feature: the food entry form. We had originally planned to use information from the freefood listserv as our only source for free food, but users were interested in a form directly linked with the map that would allow them to provide and find food in the same place and potentially abandon the freefood listserv altogether.

Additionally, we received feedback that our scraper was missing foods. We lacked a comprehensive list of all possible food items, and we could not reasonably update the list by hand. Instead, we made our scraper more robust by having it identify when several items are in a list and assume that if one is food, the rest are likely food as well. This allows new food items to enter the database automatically.

In addition to missing some foods, we realized that our beta implementation of populating the map did not support multiple food offerings in the same location. We updated our info window and sidebar to show multiple entries, making our map more accurate and useful.

When we saw how valuable user feedback was, we decided to add a Google Analytics object to our map in order to track use and gather statistics that could help us make future decisions for FoodMap. For example, we were able to track days of heavy use, such as Communiversity, and use that to plan out when to send advertising blast emails and posts. In the future, we plan to use demographic information in conjunction with our valuable user feedback to make FoodMap even better.

Different Directions

At the start of the project, we made a somewhat unconventional choice to use two backend platforms: Node.js and Django. As stated earlier, the former was used for the email scraper, and the latter was used for the web app. We split it this way because while Ryan had experience with email scraping in Node.js, but we felt Django would be a better framework for interacting with the database, so they were useful in conjunction.

We were not initially concerned that this would pose issues, since we did not anticipate the two platforms interacting directly. (Our initial design had them interact indirectly through a well-defined interface – the database – where Node.js would write to it and Django would read from it.) However, our plans had to change after our users requested the ability to enter food emails directly into the map rather than send them to the freefood listserv first. We decided that the most convenient way to implement this for our users was to allow them to write a description of their event, and to parse out the foods on the backend ourselves, so the foods can be presented in summary information on the map. This required the Django app to directly call our Node.js code for parsing foods from a text, which posed an interface issue. Although we found a solution, in retrospect it would have been more elegant to have both sides on the same platform from the beginning – either all Node.js or all Django/Python.

Future Directions

If we had more time, we would like to make the scraper more robust by using natural language processing to detect foods. This means that if an item is clearly a location (according to the natural language processing), we can scrape it as a location. If an item is clearly food, we can add it to the list of food. This would have made fuzzy matching easier, accounting for minor errors in users' descriptions. This is already feasible, but allowing for a few typos presents difficulties, especially for words very similar to food. For this reason, it was deemed more of a hazard than a help without natural language processing.

A second improvement of interest is automatically emailing or texting users when their desired food appears. The current freefood listserv will email users whenever any food appears, so users will receive numerous unwanted emails before they receive one they want. By filtering out unwanted spam in search of only the most relevant food, we could help users find their desired food without even forcing them to check FoodMap regularly. In particular, we could filter for a

specific food like sushi, a type of food like dessert, or dietary restrictions like kosher or gluten-free.

An extension of this improvement for club leaders is to gather statistical data on the types of foods users tend to like. Club leaders are interested in providing food that will attract more members to try their club, and by providing anonymized data (with users' consent) relevant to popular food options, we could help them choose the right food. We could allow users to take a brief survey on their major interests to help customize club leaders' offerings for their demographic.

A third update is to expand to other campuses. Our code is quite modular, so with just a few changes, it could be adapted for a different campus. This would significantly increase the number of potential users, vastly increasing the number of people helped by our app. Additionally, should a user become excited by the potential latent within our code, he or she could access it and update it beyond its original state to include new and more exciting features. These improvements could then be exported to other campuses and affect all consumers.

A fourth improvement would be to have markers specialized for the type of food available. This would mean that a chicken freefood event would appear with a chicken marker, and a fish freefood event would appear with a fish marker. This would allow users to more rapidly determine what freefood events they most want to visit.

One last update to our app is scraping listservs other than the freefood listserv. Numerous events include free food but do not appear on the freefood listserv because the "free food" requires participation in their event. Our map could take this into consideration by using a different marker to specify that the food is not truly free food but rather food with an associated event. This would allow users to have more options for free food without corrupting the intention behind free food.