Harihar Subramanyam
Tawanda Zimuto
Ryan Chipman
Danielle Man

11 November, 2014
6.170 Final Project
Design Document

menyou

the menu for you

Revisions

---

1. Changed “meal” into “dish”
2. Clearer statement of purpose/refinements
3. Introduced concept of Keyword
4. Introduced concept of Mapping
5. Clearly defined what the Question concept is
6. Add refinements to recommendation algorithm
7. Many changes to data model
    a. Add Keyword set
    b. Remove Taste set from data model
    c. Changed textual constraints
    d. Reformatted data model
    e. Updated Question on data model
    f. Added Mapping to data model
    g. Turned Answer into a superset
8. Created contours on data model
9. Created MongoDB collections from contours on data model (i.e. wrote the Data Design section)
10. Added “What is a Question?” to challenges
11. Added “Locu is Slow and Limited” to challenges
12. Added “Turning Mapping into Keywords” to challenges
13. Added “Do Questions and Mappings Live in the Database?” to challenges
14. Added “No Dishes, Menus, or Restaurants in Data Design” to challlenges

Overview

Motivation

Introduction

Menyou was created out of an interest in improving the current dining experience for consumers. There are many different platforms out there that help a user decide where to eat, facilitate the discovery of new venues, and aggregate reviews and ratings for comparison purposes. The problem is that sometimes users

don't actually know what to eat. Popular apps for this task today, namely Yelp and Foursquare, simply provide a long list of restaurants and leave the user to decide which one to go to - they pay no attention to the user's taste in food or to the menus at each restaurant. Menyou is different from these apps in that it recommends a **dish**, not a restaurant. Menyou analyzes a user's taste profile while simultaneously reading all of the dishes that are close to that user, thus providing the user with the best possible dish to suit his or her own preferences.

Purpose

At its core, the purpose of menyou is:

"Recommend dishes that the user will enjoy"

That is, we want to suggest dishes at restaurants which the user will enjoy eating based on their unique taste.

However, let us add two refinements to this purpose in order to flesh it out and make our concept development easier. The first refinement is:

"Do not violate a user's dietary restrictions"

We aim to avoid recommending foods that the user does not like to eat or cannot eat. For instance, if the user is a vegetarian, we do not want to recommend a hamburger. If the user is allergic to dairy, we do not want to recommend them a milkshake. If the user hates eggplant, then we won't recommend the eggplant parmesan. Et cetera.

Our second refinement is:

"Only recommend dishes near the user"

We will search for restaurants near (within tens of miles) of the user, because these are likely to be the ones that the user will actually visit.

Context Diagram

Since interaction with many external systems is complicated and fragile, we keep menyou's context simple. The context diagram is shown below:

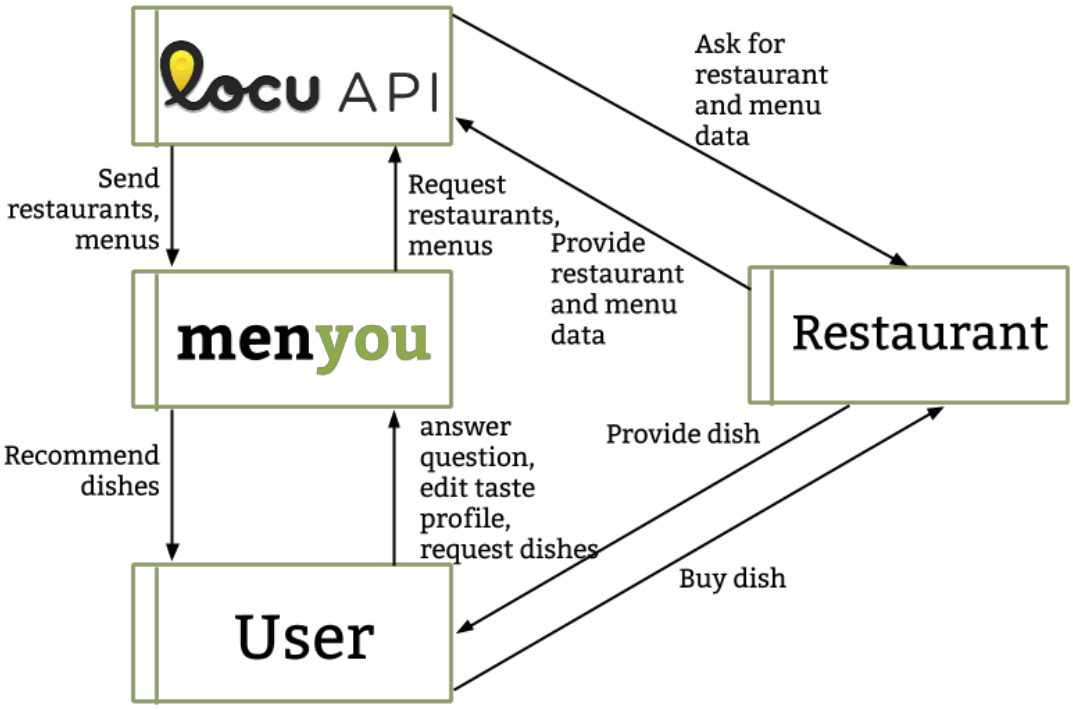


Figure 1: Context diagram

In order to keep the diagram focused and clear, we have omitted interactions and systems that are inconsequential. For instance, the user and restaurant may process payments with the help of a bank, but including the bank would add complexity to the diagram without providing us with useful insights to build menyou.

There are four systems of interest here. Ultimately, we (menyou) care about the interaction between the user and the restaurant, because the user wants a dish and the restaurant provides dishes. However, we cannot collect menu data from the restaurants directly. Therefore, we acquire restaurant and menu information from the Locu API. So, menyou asks for restaurant/menu data and Locu provides the data. The user provides their taste to menyou by answering questions which menyou uses to build a taste profile. Then, menyou recommends dishes to the user.

Note that we do not control the interaction between the user and restaurant, nor do we control the interaction between the restaurant and Locu. This is not ideal. For instance, if some restaurants are inadequately represented, or not represented at all, on Locu, we will not know. In addition, we cannot observe the user at the restaurant and gauge their enjoyment of the dish. However, given our time limit and resources, trusting Locu and using questions to determine a user's taste profile is a reasonable decision.

Design

Concepts

We shall take a top down approach in discussing the concepts - we begin with the most important concepts first.

dish recommendation - This concept is ultimately what achieves menyou's purpose. It consists of a dish and the restaurant that serves the dish (ex. "Try vegetarian pad thai at Pepper Sky's").

dish - A purchasable food served at a restaurant. It consists of a name, a description (ex. "Char-grilled chicken served shish-kebob style"), and a price. Since the purpose of the app is to recommend dishes, we need this concept to define what a dish actually is.

menu - A list of dishes. The Locu API provides us with menus, from which we extract dishes in order to recommend them. Our purpose is to recommend dishes, and menus are the containers for dishes.

restaurant - A venue that serves food. It consists of a name, an address, and a geographic location. Our purpose requires us to recommend dishes **near the user**, so we introduce this concept to ensure that recommended dishes are at nearby restaurants.

keyword - A string representing an aspect of a meal. For instance, possible keywords include "spicy", "eggplant", "deep-fried", "ice cream", "Cajun", and more. We introduce this concept because a user's taste can be represented as lists of keywords and because our recommendation algorithm will search for keywords when deciding which dishes to recommend.

mapping - A keyword and an associated set of other keywords. This is used to denote a large set of keywords using a short name. For instance, "shellfish" could map to "lobster", "clam", "mussel", "shrimp", "crab", "oyster", etc. This way, a user can refer to a large set of food with a single keyword. With this, we can make it easier for the user to specify their taste profile, and therefore allow us to make better recommendations.

taste profile - Our purpose is to recommend dishes that the user will **enjoy** while respecting their **dietary restrictions**. Therefore, we introduce the concept of a taste profile, which captures the following aspects of a user's taste:

1) likes - A list of keywords representing aspects of a meal that that a user likes
2) dislikes - A list of keywords reflecting aspects of meal that a user dislikes.
3) forbidden - A list of keywords reflecting aspects of a meal that a user will NOT tolerate (ex. if the user is vegetarian, then "beef" would be forbidden)

The taste profile captures a user's taste. Thus, it can help us identify dishes that the user will enjoy, which means helps us meet our purpose of recommending dishes.

question - An interrogative statement posed to the user to help build their taste profile. It takes the form "What are your thoughts on <keyword>?", and it has three possible responses: "I like it", "I don't like it", and "I hate it". Based on the answer to the question, we can add it to the likes, dislikes, or forbidden list of the user's taste profile. By utilizing the concept of questions, we allow users to add to their taste profile, thereby improving ability to recommend dishes the user enjoys.

Recommendation

In order to understand the data model for menyou, we must first understand the recommendation algorithm. While recommendation is typically done using machine learning techniques such as collaborative filtering, we will not be adopting that approach because setting up and tuning such an algorithm would be too time consuming (and we don't have a great deal of time). Therefore, we shall adopt a simpler approach that involves filtering out forbidden dishes, and assigning (or subtracting) points to liked and disliked dishes.

Input

The input to our algorithm will be a **menu** (a list of dishes) and a **taste profile** (the keywords representing the user's likes, dislikes, and forbidden keywords).

Denote the menu by M. Let M[i] be the ith dish on the menu. Let M[i].name, M[i].description, M[i].price, M[i].restaurant, M[i].address, and M[i].location be the name, description, price, restaurant name, restaurant address, and restaurant latitude/longitude of the ith meal on the menu.

Denote the taste profile by T. Let T.likes, T.dislikes, and T.forbidden be the lists of keyword. For instance, if a user likes eggplant, then "eggplant" would be an element in T.likes. If a user absolutely will not eat dairy, then "milk", "yogurt", and "cheese" would be elements of T.forbidden.

As a demonstrative example, consider the following values for M and T.

```
M = [{
        "price": 10,
        "name": "Eggplant parmesan",
        "restaurant": "Antico Forno",
        "address": "123 Something Way, Somewhere, AK, 12345"
        "location": [42.12312, -71.2342],
        "description": "Breaded fried eggplant with marinara sauce and parmesan cheese, sprinkled with
        basil"
},
{
        "price": 5,
        "name": "Spring rolls",
        "restaurant": "Thai 123",
        "address": "125 Something Way, Somewhere, AK, 12345"
        "location": [42.312, -71.22342],
        "description": "Fried pastry rolls filled with cabbage and carrot. Served with
        hoisin dipping sauce"
}]

T = {
        "likes": ["eggplant", "cabbage", "carrot", "pastry"],
        "dislikes": ["hoisin"],
        "forbidden": ["cheese", "milk", "yogurt"]
}
```

Output

The output of the recommendation algorithm will be a list of recommended dishes, call it R. The list R will be a subset of M, but each meal will be augmented with an attribute "points", where the higher the points are, the more highly recommended the meal is.

Algorithm

The pseudocode for the algorithm is described below:

1) Let R be a new list

2) For each meal, m, in M
    a) Let points = 0
    b) For each forbidden, f, in T.forbidden
        i) If (m.description + m.name) contains f (case insensitive)
            1) skip to the next iteration of the for loop on line 2
    c) For each like, e, in T.likes
        i) If (m.description + m.name) contains e (case insensitive)
            1) points = points + 1
    d) For each dislike, d, in T.dislikes
        i) If (m.description + m.name) contains d (case insensitive)
            1) points = points - 1
    e) m.points = points
    f) Add m to R
3) Return R

The algorithm is quite straightforward. It simply searches in m's name and description for each of the strings in T.likes, T.dislikes, and T.forbidden. If it finds a forbidden keyword, then it skips m and does not add m to the list R. For each keyword in T.likes that is found in m's name or description, points is incremented. Similarly, for each keyword in T.dislikes that is found in m's name or description, points is decremented. Finally, m is augmented with the "points" attribute and added to R. After all the dishes in M are processed this way, the list R is returned.

Running this algorithm on the example input above, we would get:

```
R = [{
        "price": 5,
        "name": "Spring rolls",
        "restaurant": "Thai 123",
        "address": "125 Something Way, Somewhere, AK, 12345"
        "location": [42.312, -71.22342],
        "description": "Fried pastry rolls filled with cabbage and carrot. Served with
        hoisin dipping sauce"
        "points": 2
}]
```

Notice that eggplant parmesan is not include in R because the taste profile indicates that "cheese" is forbidden. The spring rolls get 3 points for "cabbage", "carrot", and "pastry", but they lose 1 point for having "hoisin" - the net total is 2 points.

Notice that the price is not used in this algorithm, because filtering by price is handled on the client side (i.e. the user selects a price range and client hides the recommendations which fall outside the price range). Similarly, the address, restaurant, and location are not used in the algorithm, but are passed to the client for displaying the restaurant on a map.

We have made some refinements to the recommendation algorithm to improve the quality of recommendations, they are:

1. Use fuzzy string matching instead of exact matching (ex. "pineapples" should match "pineapple").

2. Do not recommend the same meal from a restaurant twice (sometimes restaurants have duplicates dishes in the Locu API)
3. Only recommend at most 3 dishes from any restaurant. This prevents the situation where all our recommended dishes come from just a few restaurants

Now that we understand the algorithm, let us turn our attention to the data model.

Data Model

The data model is displayed below:

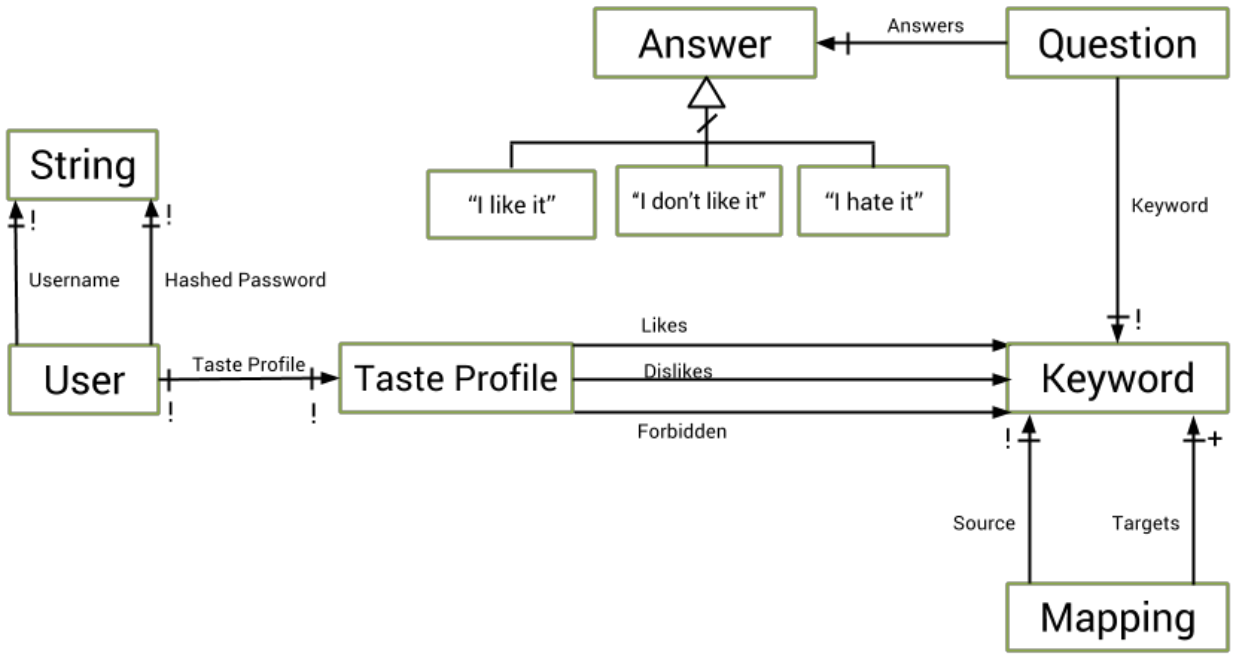


Figure 2: Data Model

The data model has been simplified a great deal from our previous version, but captures more information.

Let us consider each part of the data model.

A User has exactly one username and password (which cannot be changed).

Each User has exactly one Taste Profile. The taste profile consists of a set of Likes, Dislikes, and Forbidden Keywords. Each of these sets can be mutated.

A Keyword is simply a string representing one aspect of taste.

A Mapping consists of exactly one source Keyword (ex. "shellfish") and at least one target Keyword (ex. "lobster", "clam", "mussel", "shrimp", "crab", "oyster"). This allows us to map a Keyword to a larger set of

Keywords. For instance, we can map “shellfish” to “lobster”, “clam”, “mussel”, “shrimp”, “crab”, “oyster”. Mappings are immutable.

A Question consists of exactly one a Keyword (“What are your thoughts on <Keyword>?”) and have exactly three answers (“I like it”, “I don’t like it”, and “I hate it”). Questions are immutable.

Data Design

We draw the following contours around our data model to create three collections.

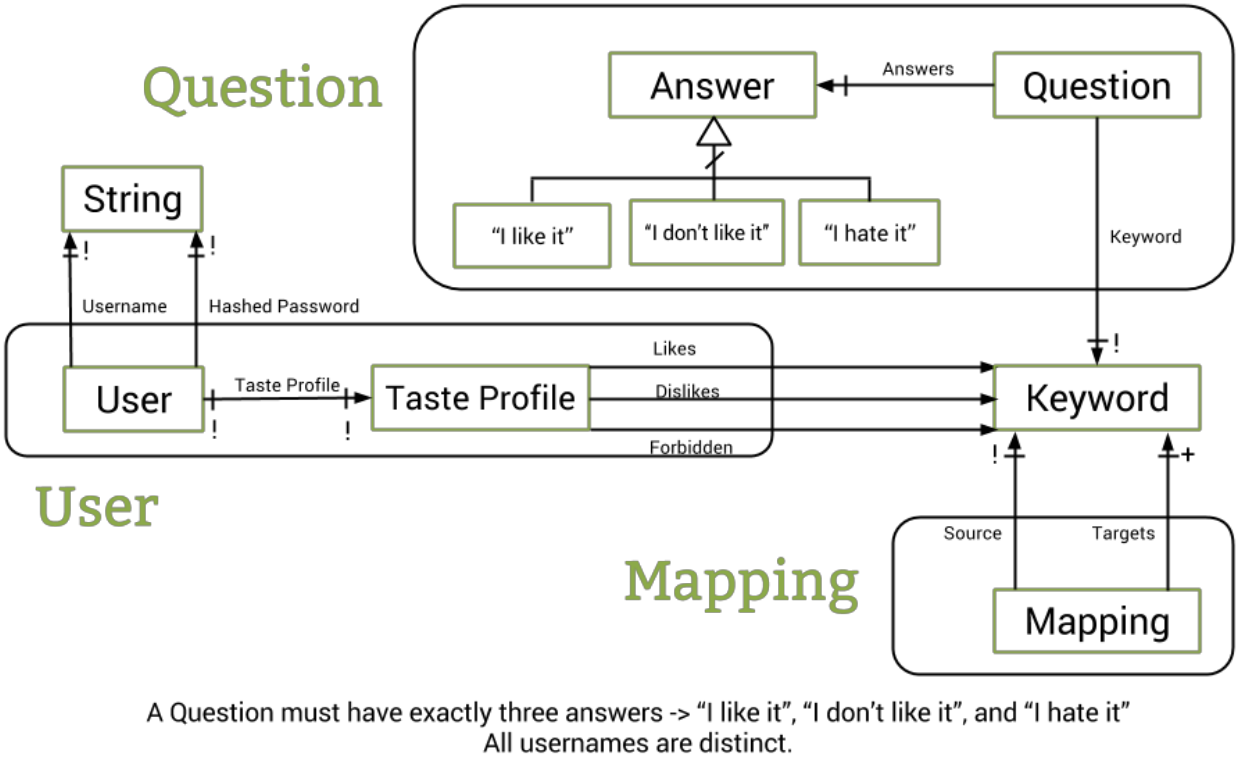


Figure 3: Contours on Data Model

From this, we can create schemas for MongoDB collections:

User:
{
“username”: String,
“hash_password”: String,
“taste_profile”: {
“likes”: [String],
“dislikes”: [String],
“forbidden”: [String]
}
}

The User collection stores the username and hashed password for a given user. Since a user is associated with exactly one taste profile, and each taste profile is associated with exactly one user, we can store a user's taste profile in the User document itself. A taste profile has three arrays - one for the likes, one for dislikes, and one for the forbidden keywords. Since keywords are effectively strings, we need not create a special collection or MongoDB entity for them.

Mapping:
{
  "source": String,
  "targets": [String]
}

Recall that a mapping associates a source keyword with some target keywords. Therefore, we have a string for the source and an array of strings for the targets.

Question
{
  "keyword": String
}

Recall that a question takes the form "What are your thoughts on <Keyword>?", and has three answers: "I like it", "I don't like it", and "I hate it". Since the answers are fixed, and the structure of the question is fixed, the only thing that we must store in the MongoDB collection is the keyword.

Behavior

Security Concerns

Security Requirements

- An authenticated user can only edit his/her own profile details & taste profile.
- Users cannot see any details about other users, including names, emails, and taste profiles.

Potential Risks

- Hackers who break into the system to extract personal information
- Spam advertisers gaining access and inserting incorrect/irrelevant information

Threat Model

- We can assume that there is minimal interest for hackers retrieving profile information, because there are much easier ways to aggregate name/email information than hacking into menyou.
- Menyou involves no interaction between users, so there is no risk of manipulation/fraud between users within the system.
- There is no payment component to Menyou, so we don't need to worry about that getting hacked.
- Spammers are the most likely of our threats.

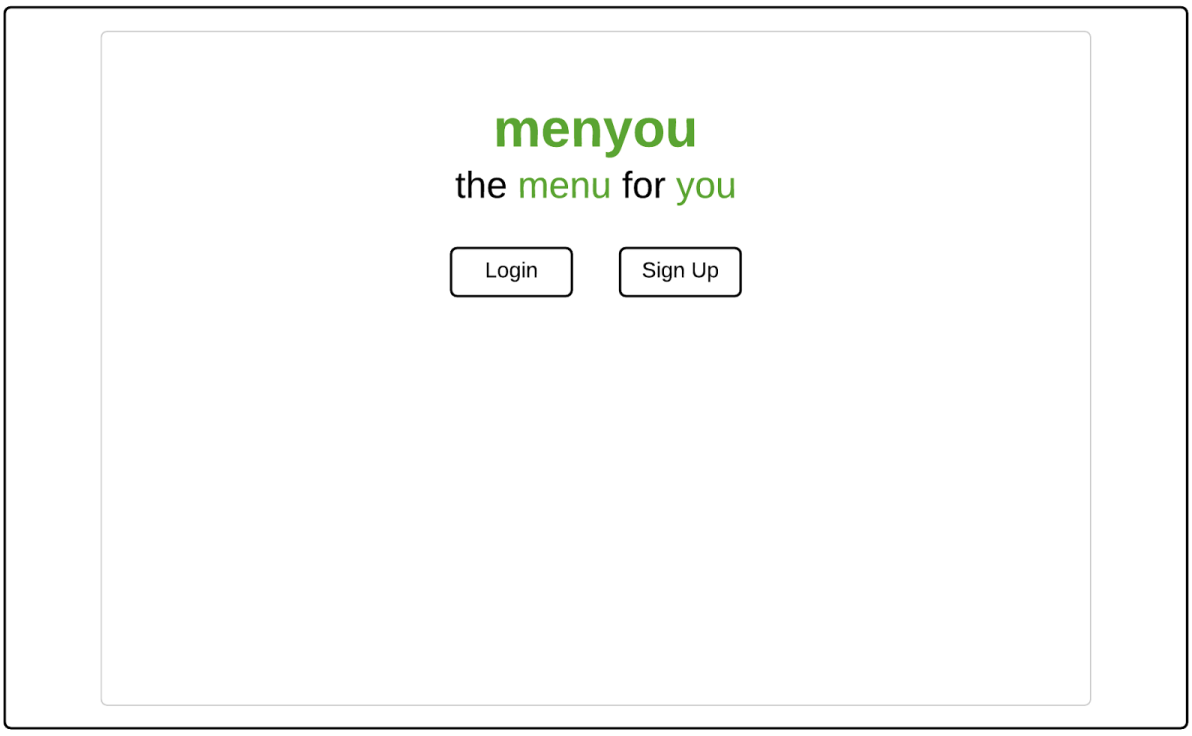
Mitigations

- Profile Information

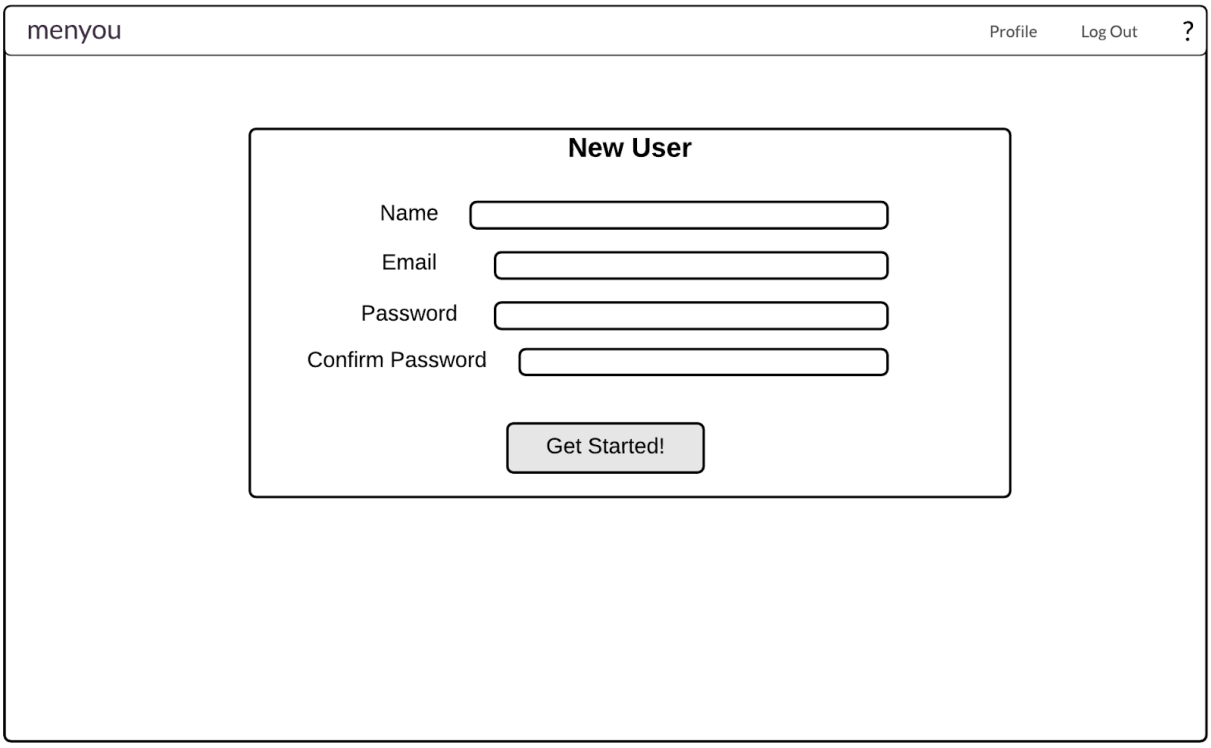
    - Properly implement access control
    - Require access token with every request. Don't use cookies to prevent session hijacking.
- Spammers
    - Trust Locu to have accurately maintained data
    - Allow users to 'report' inaccurate/spam entries
    - Standard countermeasures against injection, XSRF, etc.

User Interface

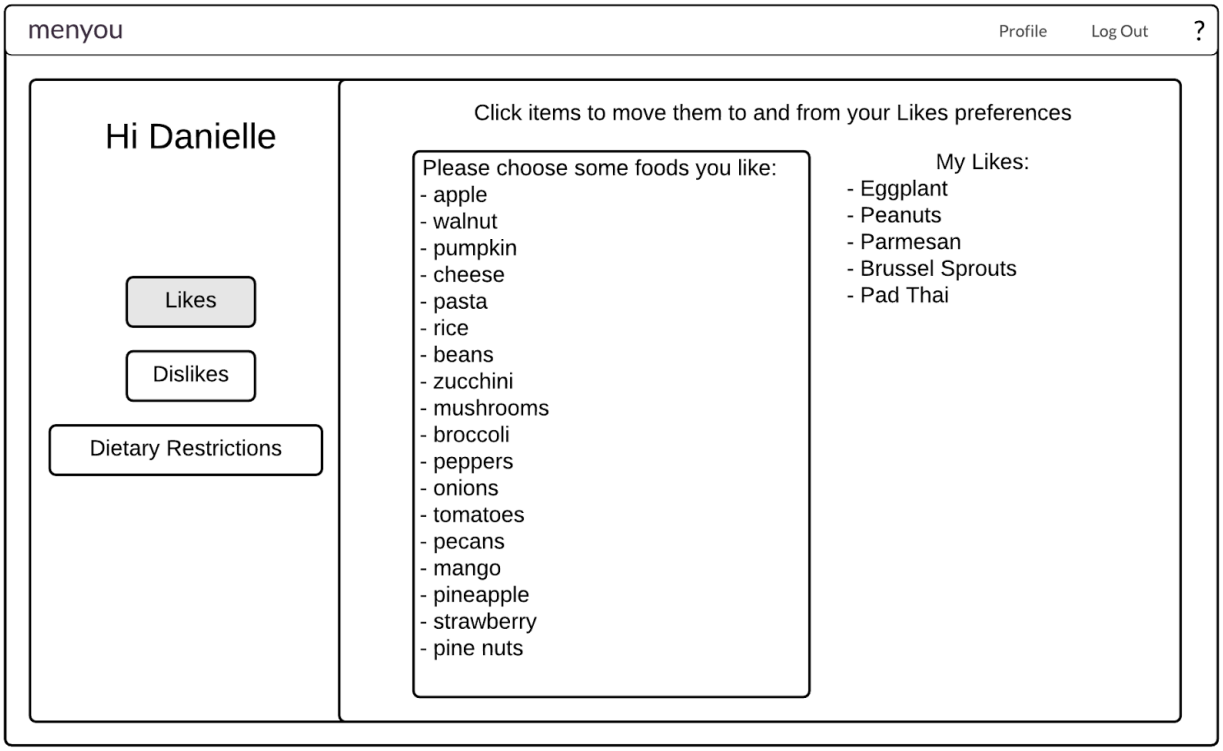
User not logged in:



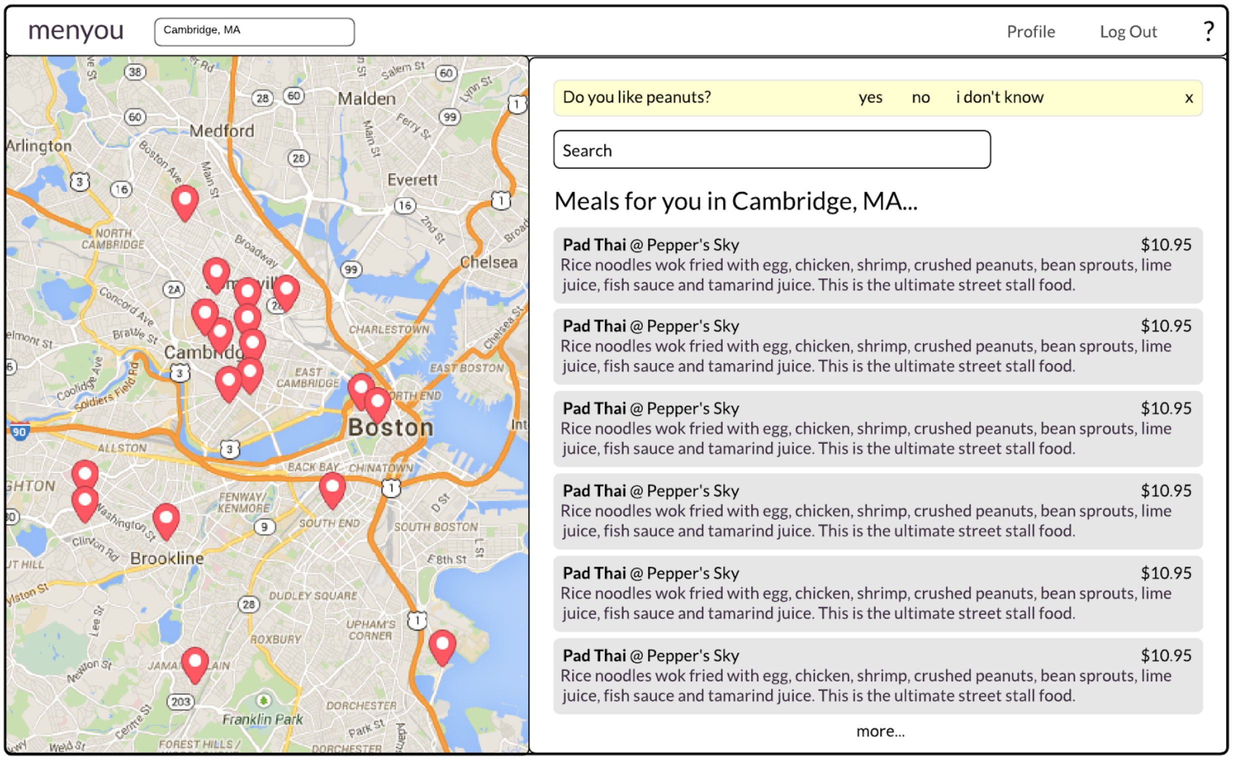
User Sign Up page:



User Profile page:



User Home page:



Challenges

Concepts

Taste Profile Refinement

Problem Description

Our taste profile will by no means be perfect after the user's initial specification of their likes/dislikes. In order to provide consistently satisfying recommendations, menyou needs to incorporate a feedback loop for improving the quality of the recommendations by augmenting the taste profile.

Potential Solutions

- Dedicated Profile Editor
- Infer Preferences
- Questions

Our Decision

We decided to include both the first and third options in our design. The fine-grained control provided by a dedicated profile editor is useful and worth including, but having an incremental way to add to the taste profile (ex. by asking the user a simple question visible to them while they use the app) we are able to get data to improve a user's taste profile more often.

Focusing on Dishes

Problem Description

One of menyou's key goals is to change a user's dining decision process so that the focus falls on choosing a good meal (rather than choosing a restaurant). That being said, information about the restaurant is still important and needs to be provided to the user, even though our *primary* focus may be dishes. Our challenge is to find a way to make all the information easily accessible to the user without allowing the secondary/extraneous information to overshadow the individual dishes.

Potential Solutions

- Full-Page Map
- Full-Page Dishes List
- Dishes by Restaurant
- Meal List + Restaurant Map

Our Decision

We will be implementing the third of the three options discussed above. It is the only of the three that actually allows the user to see his best meal options without obscuring the secondary (but still important) information like location. Furthermore, it provides that additional information in a simple, easily digestible visual format.

Data Design

What is a Question?

Problem Description

A question is an interrogative statement posed to the user to understand some aspect of their taste. This is very vague. What will a question look like? How will we generate them? How will the user respond? How will we learn from the user's answer? To resolve this, we must clearly define what a question is and what the answers look like. Furthermore, we need to consider how this decision will affect the representation of a question in our data design.

Potential Solutions

- Handwrite questions, answers, and logic for updating taste profile
- Create a set of question templates and generate questions, have mutliple methods of updating taste profile.
- Use one question template, one method for updating the taste profile..

Our Decision

We decided to go with the third approach. Our questions will take the form "What are your thoughts on <Keyword>?", where <Keyword> is randomly selected from a handwritten list of keywords (we have already created this list). The responses to the question will be "I like it" (if the user selects this, <Keyword> will be added to their likes), "I don't like it" (if the user selects this, <Keyword> will be added to the dislikes), and "I hate it" (if the user selects this, <Keyword> will be added to their forbidden list). There are a number of reasons we chose this approach. First, it minimizes the amount of question/answers/code we have to write. Second, it provides a uniform format for questions which will make it easier to design the UI to display the questions. Third, by having a consistent and simple format for the questions, it will be clearer to the user how the question system works, and they'll know how to best use it to build their taste profile. Finally, since a Question can be constructed from single keyword, it is very simple to represent in our database - each Question is simply represented by a keyword.

Do Questions and Mappings Live in the Database?

Problem Description

We've displayed the Questions and Mappings in the data model, and we've shown how they can be mapped to MongoDB collections. However, the collections are quite simple (Mapping contains just two fields, and Question is just a single field). Do these really need to be part of the data design? Why store them in the database at all? Why not store them each in a static file which is read when the server starts?

Potential Solutions

- Store Mappings and Questions in flat files which are read on server start
- Store Mappings and Questions in database, but load them in main memory on server start
- Store Mappings and Questions in database, and load them in main memory as needed

Our Decision

Firstly, since the Questions and Mappings are entity sets which help represent the data that is important to our app, we have represented them in the data model. As for storing them, we decided to go with the second option. Although we do not anticipate the number of Mappings and Questions to become enormous (after all, we must write them), there are a number of advantages in storing them in the database. For instance, we can find the source word associated with a given target (ex. "oyster" is a type of "shellfish"). However, for efficiency purposes, we can store them in main memory when the server starts so that no database access (a slow operation if data is read from disk) is needed to serve them to the user.

No Dishes, Menus, or Restaurants in Data Design

Problem Description

We highlight the concepts of meals, menus, and restaurants, but we do not make any mention of them in the data design. Should we consider them in the data design?

Potential Solutions

- Yes
- No

Our Decision

We decided not to consider dishes, menus, and restaurants in the data design. This is because we don't have direct access to the raw data - we must fetch it through the Locu API. While there may be advantages to having that data in our database, Locu prevents us from making enough API calls to get a substantial dataset that we can put in our database. Furthermore, adding code to fetch data from Locu and store it in our database adds complexity to the project. Thus, even though getting the data from Locu and storing it in our database would be nice, the API does not allow it, and it could be time consuming.

Code Design

Turning a Mapping into Keywords

Problem Description

We introduced the concept of a mapping, which associates a source keyword (ex. "shellfish"), with several target keywords (ex. "lobster", "clam", "mussel", "shrimp", "crab", "oyster"). With this concept, the user is able refer to a larger class of foods using just one keyword. The taste profile, however, only consists of keywords, so how will the mappings fit into the picture?

Potential Solutions

- Modify taste profile to support keywords and mappings
- Store target keywords (and source keyword) in taste profile.

Our Decision

We decided to take the second approach. So, if the user marks "shellfish" as a forbidden keyword, the word "shellfish", as well as the words "lobster", "clam", "mussel", "shrimp", "crab", and "oyster" will be added to the user's forbidden list in the taste profile. The advantage in this approach is that we do not need to modify our recommendation algorithm or database at all to support this. The disadvantage is that, when a user wants to remove a particular mapping from their taste profile, they will have to remove each keyword. However, since mappings tend to be small (each source will map to approx 3 - 8 words), this will not be a great deal of effort. Furthermore, by making the taste profile consist only of keywords, it will be easier for the user to understand and manage. Finally, it is unlikely that the user's taste will change often, so the situation where the user needs to modify his/her taste profile and remove a number of mappings is rare.

Authentication

Problem Description

Our authentication scheme needs to satisfy two key criteria. First, since we are providing menyou as an API backend, we need the authentication scheme to be flexible and adaptable to many kinds of clients. Second, it needs to provide adequate protection against the security risks outlined earlier in the document.

Potential Solutions

- OAuth
- API Tokens
- Username/Password + Cookies

Our Decision

We decided to implement authentication using the second of the three schemes. Except for the new-user route (which requires no authentication) and the login route (which takes username/passwd in the request body), every API call expects a bearer-token to be provided with the Authentication request header. OAuth was overkill for our particular use-case, but a token-based implementation provides major conveniences in the form of cross-platform compatibility and easy session invalidation (invalidating a user's sessions requires only generating a new API token). Cookies and other such conveniences will be implemented client-side; the backend will use the same implementation regardless of client platform.

Scoring Dishes

Problem Description

In order to provide menyou's users with recommendations, we need to have some method of ranking dishes. Given a taste profile and a menu, we need to produce a nonnegative integer that represents how strongly we recommend that meal for that given user. Distilling such subjective factors as food preferences into a single integer requires some careful consideration about what makes a good recommendation.

Potential Solutions

- Machine Learning (ex. collaborative filtering)
- Recommendations from other users
- Point/filter based recommendation algorithm

Our Decision

We decided to go with the third option. Machine learning algorithms would be complicated to set up and integrate in the app given our limited time, so we chose to avoid them. Getting recommendations from other users would also require a good deal of code to be implemented to handle building a relationship between users and a way for users to recommend food to each other. However, recommendations from other users may not be very useful, because people have very different tastes, and a recommendation from somebody else may not fit your tastes just right.

Locu is Slow and Limited

Problem Description

Since we have a free plan with Locu (instead of a paid plan), Locu is slow to respond to our requests. It takes about 3 seconds to respond to our request and it only sends us 25 restaurants for each request. To make useful recommendations, we'd like to get more than 25 restaurants, and we'd like to make sure that user is not confused if it takes a few seconds to get recommendatinos.

Potential Solutions

- Make multiple requests in parallel
- Display an appealing "Cooking up recommendations" loading screen
- Try to fetch recommendations before user asks for them, cache recommendations

Our Decision

We decided to go with the first and second option. In order to get more than 25 restaurants, we make one request, and then we select three random likes from the user's taste profile and search for restaurants matching those keywords. We make these requests in parallel. This way, we get up to 100 restaurants returned (multiple requests), we ensure that we get some meals that the user will like (searching for likes), and we ensure that we don't take much additional time (parallel requests). We decided not to try to preemptively fetch recommendations because the user must see recommendations when they load the app, so we'll have to fetch recommendations when the user loads the app. We decided not to cache recommendations because the added challenges of keeping a cache for each user, updating it as the taste profile changes, and invalidating it periodically would a great deal of complexity to the codebase and might actually degrade user experience (ex. a user is frustrated that he/she sees the same recommendations). So, we decided that we will create an appealing loading screen (ex. display "Cooking up recommendations" with a GIF of a cartoon chef cooking up some food) to entertain the user for the 2-3 seconds where the recommendations are fetched. Of course, the most straightforward solution to this problem is to upgrade to a paid plan with Locu, but that costs money. If we extend menyou after 6.170, we will look into buying a plan from Locu.

Displaying Questions

Problem Description

We refine a user's taste profile by asking them small, simple questions. In order for this method to be successful, people need to actually answer the questions. To make this happen, it is essential that our front-end implementation of these questions strikes a balance between prominence (so that a user actually sees the questions we want them to answer) and unobtrusiveness (so that we don't annoy the user and cause them to perceive our questions as 'spammy').

Potential Solutions

Modals/Pop-Ups

In list of menu items

Our Decision

We shall display the small, simple questions at the bottom of the menu list. This way, we do not interrupt the user as we would with a modal/pop-up, and we ensure that the question is always onscreen so that the user can respond to it.