Phase One Technical Report

Syscall of the wild

Omar Lalani, Alejandro Landaverde, Mike Wham, Kevin Wheeler, Nathan Heskia

Table of Contents

Introduction

Problem

Feed.ly is designed to help people decide and find what they want to eat. When posed with the question, “Hey. What do you want to eat?”, why is it that most people respond with the name of a restaurant or a cuisine instead of the actual dish that they want to eat? Lets face it, you have a craving and nothing else will satisfy it. You may want a chicken quesadilla but you respond, “Lets eat Mexican.” You may want some gourmet pizza but you say, “Lets eat Italian.” Next, you start looking up restaurants that serve this cuisine and read their reviews. But the problem is that nobody writes reviews for dishes, they write them for restaurants. Sure, this is helpful in some cases if you want to know how long food takes on average or how friendly the wait staff is; but, you aren’t going to eat the restaurant. You’re going to eat the dish. This is what feed.ly aims to change. A user can look up “alfredo pasta” and get a listing of every restaurant in his or her area that either has this dish or something similar. Additionally, dish revies are also available. Feed.ly enables users to not only find the closest enchilada, but the best enchilada. The content of feed.ly will consist of three resources:

1) Dishes:

    Every dish has a page with its description and on this page will be reviews for the specific dish as well as links to similar dishes from this and other restaurants.

2) Restaurants:

    Every restaurant has a page listing its location, hours of operation, and a link to its company web page along with other information about the

restaurant. This page also has reviews about the restaurant and a list of dishes available at the restaurant with ratings.

3) Reviewers:

Every reviewer has a page with information about the reviewer. This page will also have a list of every restaurant the reviewer has reviewed/rated as well as every dish the reviewer has reviewed/rated.

Use Cases

1. **"Hmmm…what's the highest rated dish in Austin?"**

Our front landing page will house a list that tracks the highest rated dishes in the Austin area allowing the end user to quickly read reviews for these dishes and decide what and where they want to eat.

2. **"Its 2 AM on a Thursday and I can't sleep unless I eat a really good burger"**

Our end user will be able to type in "burger" and get a listing of the top burgers in his or her area along with restaurants that serve burgers. Additionally, the list will display places that are open based on the times we have in our database for the restaurants.

3. **"That's how that particular reviewer rated the dish, but I'm a lot more critical…how do I know I will like the dish?"**

By analyzing ratings and reviews we will guess how a particular reviewer would rate a dish they have never eaten before. This algorithm will be similar to the Netflix algorithm used to rate movies for viewers that have never seen the movie.

4. **"The meal I ordered was the best meal ever…but the service sucked"**

Every reviewer has the option to review either a restaurant, a dish, or both. These ratings are kept separate from each other. This way, a user will know that a restaurant may have bad service but really good food.

Design

RESTful API

The first step in designing our API is to consider the information a user might want to get from our API. Since feed.ly is a meal review and recommendation service, we think that our resources can be categorized into restaurants, dishes(generic and specific), customers, and cuisines. These four categories are the main semantic descriptors of the feed.ly API and are used to form the API's hierarchy. These different descriptors reflect user navigation on feed.ly. For example, a user can first choose a type of cuisine, then a generic dish within that cuisine, and then a specific dish in addition to reading customer reviews. But it also makes sense for a user to find a specific dish and reviews and information about restaurants that serve similar dishes. This nested decision making is what we want to offer with our API. We want users to be able to get different representations of our four main resources: all dishes at a restaurant and/or all dishes categorized under a specific cuisine.

Since our API is read-only, our only protocol semantics are GET requests. This is our main motivation for using a JSON format for responses instead of a JSON+Collection format since we don't need to use template objects to enable writing. The response code for each request a user makes is 200 (OK) and the corresponding JSON document represents the data of the requested resource. Each category of the

feed.ly API hierarchy has a URL that enables a user to get a list of all objects in that category. For example, a request for `/api/dishes/` returns a JSON document containing a list of all specific dishes and their corresponding features (cost, vegetarian/vegan, etc.). A closer look at the JSON document response shows that the very first key contained in a dish item is `dish_id`. Since the value of this key is not a descriptive attribute, we chose to place it first for organizational purposes and have the rest of the keys map to descriptive attributes. This same format is used when getting lists of all other main categories/descriptors in the feed.ly API, and is included to enable users to read from our database easily. The URIs in each category of our API utilize the ID numbers corresponding to the database tables for each category. The template for each category enables users to easily filter resources depending on how they would like to implement the API.

There isn't a real-world hierarchy between restaurants, dishes, customers, and cuisines. These categories are dependent upon each other. When designing the API, we were conflicted about how much of each related category to include when returning data for one category. For example, if a user requests information on a certain restaurant's attributes, should he or she also get a list of all dishes, dish reviews, and restaurant reviews related to the restaurant? We decided to design our API such that each category is self-contained. Introducing data from a related category will cause us to have to modify each category significantly if we decide to add another major resource to feed.ly in the future (i.e., drinks or ingredients). Instead, we return id numbers from related categories that enable users to look up information if they need it. For example, instead of including a list of a restaurant's dishes in response to a request for a restaurant's details, users can

filter a restaurant's dishes using a restaurant ID number using the URI

`/api/restaurant/{id}/dishes`. Other categories follow a similar procedure.

API Quick Reference

Restaurants

GET	/api/restaurants	List all restaurants.
GET	/api/restaurant/{id}/attributes	List restaurant attributes.
GET	/api/restaurant/{id}/dishes	List restaurant dishes.
GET	/api/restaurant/{id}/reviews	List restaurant reviews.

Customers

GET	/api/customers	List all customers.
GET	/api/customer/{id}/attributes	List a customer's attributes.
GET	/api/customer/{id}/restaurantreviews	List a customer's restaurant reviews.
GET	/api/customer/{id}/dishreviews	List a customer's dish reviews.

Dishes

GET	/api/dishes	List all dishes.
GET	/api/dish/{id}/attributes	List a dish's attributes.
GET	/api/dish/{id}/reviews	List a dish's reviews.

Generic Dishes

GET	/api/genericdishes	List all generic dishes.
GET	/api/genericdish/{id}/attributes	List a generic dish's attributes.
GET	/api/genericdish/{id}/dishes	List dishes of generic dish's type.

Cuisines

`GET`	`/api/cuisines`	List all cuisines.
`GET`	`/api/cuisine/{id}/attributes`	List a cuisine's attributes.
`GET`	`/api/cuisine/{id}/dishes`	List a cuisine's dishes
`GET`	`/api/cuisine/{id}/restaurants`	List a cuisine's restaurants.

Django Models

Our data model categorizes food in a hierarchy of three food levels: Dish, Generic Dish, and Cuisine. For example, Olive Garden's "Alfredo Pasta" is categorized as follows: the **Dish** is "Alfredo Pasta" which is categorized under the **Generic Dish** "Pasta", which is categorized under the **Cuisine** "Italian." This example leaves out an important relationship between Cuisine and Restaurant. We've decided to make a one-to-many relationship between Cuisines and Restaurants. However, we considered many alternatives. We originally considered making each Restaurant have a string value named `type_of_food`. This approach ensures flexibility, since a string variable can store a value like "Mexican with Asian Fusion and a focus on deserts." However, this approach doesn't enable good search capabilities for users. If a user wants a list of all Mexican restaurants, the only way he or she can achieve this is via a regex search. We can search for all restaurants having a substring "Mexican" somewhere in their `type_of_food` attribute, but what if a user decides to make a restaurant's `type_of_food` value "Hispanic food" instead? In this case, our regex search fails. Instead of having a string variable, each restaurant can have a foreign key that references a Cuisine table. This provides the search capabilities we are looking for. With a simple query, users can easily obtain a list of all Mexican restaurants. This approach has the disadvantage that it isn't

very flexible when restaurants don't fit neatly into cuisine types. The flexibility of having a string variable `type_of_food` is superior in this regard. For a while, we were torn between which approach to implement; but, then we realized that we could do both. We have placed a foreign key in each restaurant that references a cuisine, and also gives each restaurant a string attribute called `description`. This ensures that users have good search capabilities and that restaurants can be categorized with more flexibility.

We also had trouble finding the best way to represent the hours of operation of a restauarnt. One method of representing the data is to have variables like `Monday_open` and `Monday_closed` for each day of the week. This method is specific and handles special cases; but, more often than not, a restaurant has the same hours for Monday through Friday and special hours for Saturday and Sunday. This makes having an open and close for each day seem repetitive and unnecessary. We can represent a restaurant's hours of operation with a separate table called `Restaurant Hours` which holds a restaurant's typical weekly hours and have a one-to-many relationship from `Restaurant` to `Restaurant Hours`. For example, a row/tuple in Restaurant hours can read "Mon-Friday 8am-8pm" and another row/tuple that reads "Monday 8am-8pm, Tuesday 10am-10pm." This method allow us to store less information in our database since many restaurants share similar hours (i.e., 8 AM – 6 PM) and also give us flexibility. But this approach could complicate our data since we are dependent on many relationships for `Restaurants`. We now choose to represent a restaurant's hours of operation by having seven strings representing the store hours for each day of the week (`monday_hours`, `tuesday_hours`, etc.). This method enables us to handle special cases without having too many attributes and consuming more space than necessary.

Another set of relationships involves a Customer. A Customer has a one-to-many relationship from itself to `Dish_Review` and `Restaurant_Review`. Originally, we had a table with four attributes: `Restaurant_Review`, `Restaurant_Review_Comment`, `Dish_Review`, and `Dish_Review_Romment`. This one table had a one-to-many relationship from `Customer`, `Restaurant` and `Dish`. A customer could make a review of a dish and a restaurant at the same time. This method kept a customer's reviews concise and in one place, but required a customer to submit a review for both a dish and a restaurant, without the option of choosing between the two instead. We split `Review` into two separate tables: `Dish Review` and `Restaurant Review`. In both of these tables we have two attributes: `rating` and `review_comment`. This method enables customers to give a rating and a comment to either a dish, restaurant, or both. Our database now enables querying for just a customer's review of a restaurant, whereas the previous method had a customer reviews contain information about both a restaurant and a dish. Finally, we decided to have one `Review` table with two attributes: `review` and `comment`. This generic table enables users to review both dishes and restaurants. So if a customer wants to review a dish and a restaurant, two separate rows/tuples are created in the `Review` table for each review. This method enables us to expand customer reviews without having to change the table. We only need to ensure the foreign key coming from either restaurant or dish is filled out. But having two separate tables, `Restaurant Review` and `Dish Review` is the best way to store this information.

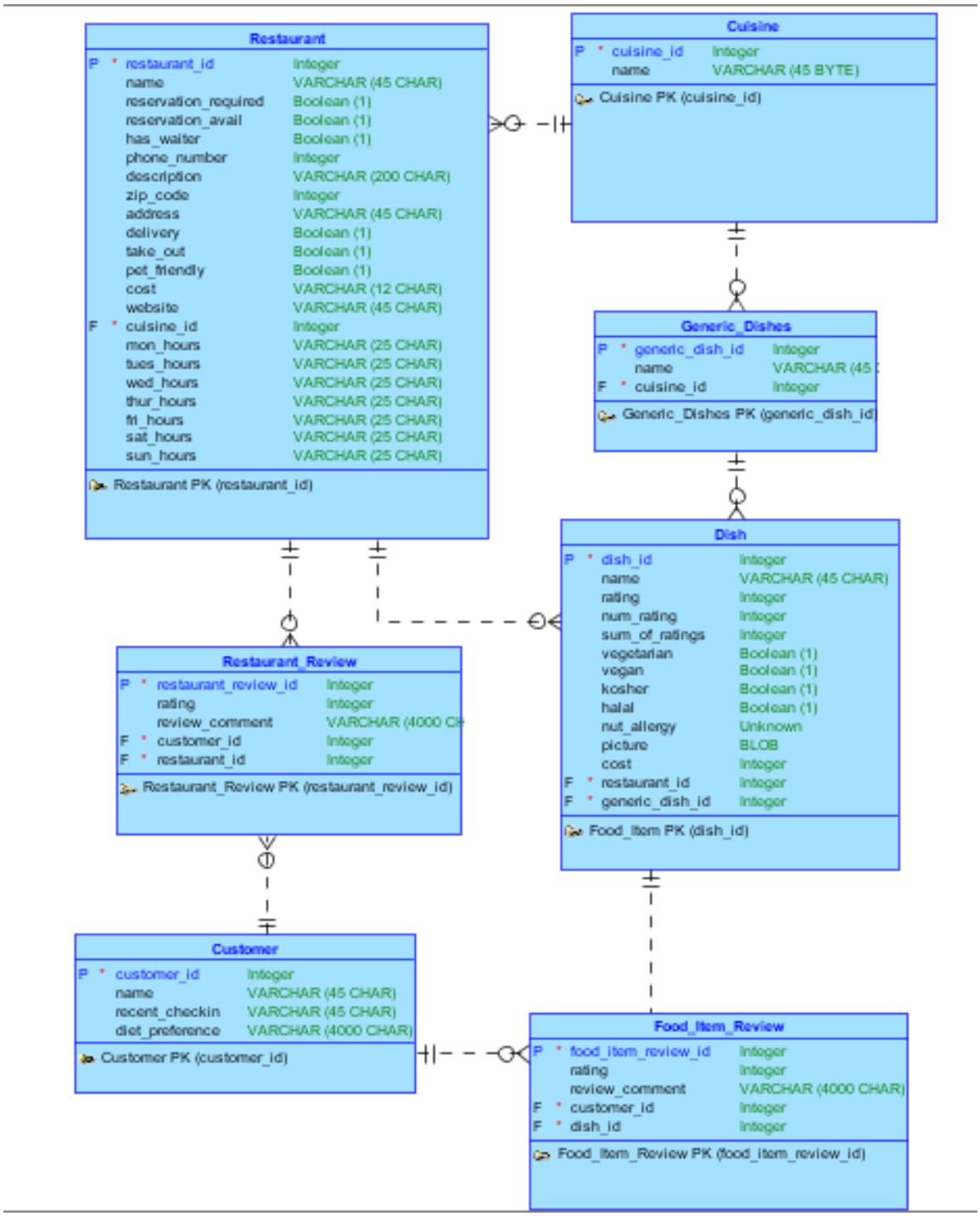
We also considered how to calculate the average rating for a particular restaurant or dish. On a restaurant or dish's web page, we display the average rating for that

restaurant or dish. Every time a customer leaves a rating for a restaurant or dish, the average rating for that restaurant or dish changes. A simple, naïve solution is to sum the ratings for a restaurant/dish after each customer review and then divide the sum by the number of ratings the restaurant/dish has. However, we don't want to go through our entire Reviews able after each new customer review. Another approach is to only recalculate the average rating for a review once a day. This is also too complicated. Yet another approach is to keep track of the current sum of all the ratings for a restaurant/dish and the current number of ratings for that restaurant/dish. If a customer leaves a rating for a restaurant/dish, we can quickly add his or her rating to the sum of ratings for that restaurant/dish and increment a counter that keeps track of the number of ratings for that restaurant/dish. The average rating is then quickly recalculated using the updated information (the sum of the ratings and the count of ratings). However, this approach introduces a possibly inconsistent database. Calculating the average rating for a restaurant/dish using the method just described may obtain a different result due to programming errors. An inconsistent database might have one value (sum of reviews attribute / num reviews attribute) for an average rating while manually going through the reviews table and summing the reviews yields a different average rating. Although this design allows for the possibility of an inconsistent database it is better than recalculating the average for a dish/restaurant by scanning an entire review table every time a customer leaves a review. We only need to ensure that we don't introduce inconsistencies through program errors.

Logical Data Model

This is a representation of our logical data model in Informational Engineering Notation.

We used Oracles free Data Modeler application for convenience.



Tests

Unit Tests

For the feed.ly Django Unit Tests we tested the `get_all()` methods that were made for each of the classes/table in our `models.py` file. The `get_all()` method returns all of the attributes of a class as dictionary. Returning a dictionary is faster when trying to display the information of a Restaurant, Customer, Dish, etc. and this will help when our HTML templates are filled out dynamically in Django. When testing the `get_all()` methods in `ourapp/test.py` file, we create instances of each class in tests for each method in order to test the function. We don't have an implementation of MySQL working for our Django app in this phase. This prevents us from properly testing objects in our database with the methods. We are able to bypass this problem by creating an instance of the classes, even though Professor Downing mentioned that our unit tests should fail. In some of the class tests, for example in the `RestaurantMethodTests`, we create an instance of the Restaurant class and don't initialize anything that is a foreign key (i.e., `cuisine_id`). If we include it in the initialization an error occurs stating that the `ForeignKey` object does not exist. This is to be expected since our database does not contain anything. By not including any `ForeignKey` objects in our initializations we avoid many errors and pass our unit tests. Once we set up a MySQL database we will pull out of the database an actual object that is stored in it and check that everything in it is properly stored. What we currently have is creating an instance of a class and then checking that instance. While we circumvent the fact that unit tests should fail at this point, once real data is stored in our database our unit tests will be vital in making sure that information is being properly stored.