Phase Two Technical Report

Syscall of the wild

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

Table of Contents

Introduction

Problem

Kravester 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 Kravester 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 reviews are available. Kravester enables users to not only find the closest enchilada, but the best enchilada. The content of Kravester consists 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 or restaurant in Austin?"**

Our splash will house two lists that track feature dishes and featured restaurants in Austin. End users can then quickly read reviews for these dishes/restaurants and decide what and where they want to eat. The featured dishes/restaurants functionally will be implemented by either enabling restaurants to pay for their dish(es) or restaurant to be featured, or by creating a list of highly reviewed dishes/restaurants that are low in cost.

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 see a page displaying the highest rated burgers in Austin along with restaurants that serve burgers. Additionally, the list will display restaurants that are open based on the times we have in our database for the restaurants. One feature of our search functionality, similar to websites like yelp.com, will enable uses to filter their results based on which restaurants are open at the time of day they are searching for a dish or to narrow results to restaurants within a zip code.

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. A possible implementation of this concept includes keeping track of a user's most frequently reviewed cuisines and generic dishes. For example, if a user often reviews Thai food/curries, then Thai restaurants and curry dishes are recommended to the user. As that user begins to respond to these food/restaurant recommendations by giving ratings, Kravestar can compute the user's deviation from the mean rating of these dishes/restaurants to determine that customer's unique offset from the mean rating of recommended dishes/restaurants in the future. Additionally, since we

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

Every reviewer has the option to review a restaurant, a dish, or both. These ratings are separate from each other. This way, a user will know that a restaurant may have bad service but really good food. This feature is implemented by having restaurant pages contain reviews written about the restaurant, and a list of dishes- each with a link to a page that has reviews of the dish and similar dishes. In future implementations of the website, this will enable users to get a feel for the quality of a restaurant's food by displaying the mean of all dish ratings or to see which dishes at a restaurant are highly rated.

5. **"What is the best restaurant serving a cuisine type in Austin, and what are the best dishes I can get at that restaurant?"**

Kravester enables filtering by cuisine type. Users can use these results to find dishes and restaurants that are highly rated, or (in future implementations) filter out dishes/restaurants with specific attributes, such restaurant location or only vegetarian options.

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 Kravester 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 Kravester API and are used to form the API's hierarchy. These different descriptors reflect user navigation on Kravester. 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. For example, a list of dishes can be categorized as being from a specific restaurant or from a specific cuisine. Or a user can simply get all dishes in the database.

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. A request for each resource can be returned in JSON format by adding `/json/` to a valid URL (see the table below for a list of valid API calls). 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 of our URLs map to a view function which determines whether or not a `Format` variable is none. If `Format` is none, then a template HTML file is used, along with a dictionary of corresponding objects from our database. If `Format` is JSON, then the query result is sent to one of seven serializer classes (corresponding to each model) in `serializers.py`. Each serializer class is designed to take the objects in the query set and covert them back into native Python datatypes. Our serializers can serialize the objects from a queryset or create instance of the original Django models given a dictionary of corresponding values. Once a queryset has been serialized, it is rendered into a JSON response class object that calls the JSON renderer function from the Django REST Framework. If a user uses the wrong id number when making an API request or if the resource he or she is looking for doesn't exist, a response code of 404 is returned, along with an error message stating that the particular resource doesn't exist. For example, if a user attempts to get a list of restaurants when there are none in our database, a dictionary with an `"Error"` key is returned, and the value of that key is `"No restaurants found."` This mirrors what happens for an HTML response, in that a user sees just the error message's text in place of a dynamically generated HMTL page.

Each category of the Kravester API hierarchy has a URL that enables a user to get a list of all objects in that category. For example, a request for `/dishes/json` 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 `id`. Since the value of this key is not a descriptive attribute but rather just the primary key from the database table, it is displayed first for organizational purposes. The rest of the keys map to descriptive attributes, but also contain some foreign key id numbers in no particular order or position, since dictionaries are unordered. For example, a dish item has a cuisine id number. This enables us to access a cuisine's attributes (it's name) when creating a dynamic response using our HTML templates. This same format is used when getting lists of all other main categories/descriptors in the Kravester 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 Kravester 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 following URL :

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

API Quick Reference

Restaurants

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

Customers

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

Dishes

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

Generic Dishes

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

Cuisines

GET	`/cuisines/json`	List all cuisines.
GET	`/cuisines/{id}/json`	List a cuisine’s attributes.
GET	`/cuisines/{id}/dishes/json`	List a cuisine’s dishes
GET	`/cuisines/{id}/restaurants/json`	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 the one-to-many relationship between Cuisine and Restaurant. We considered many alternatives. We originally considered making each Restaurant have a string variable `type_of_food`. This approach ensures flexibility because a string variable can store a value like “Mexican with Asian Fusion and a focus on deserts.” However, this approach doesn’t enable users to have good search capabilities. If a user wants a list of all Mexican restaurants, the only way he or she can achieve this is via a regular expression search. We can search for 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` attribute “Hispanic food” instead? In this case, our regular expression search fails. We also considered having a the Cuisine type be a foreign key attribute within the Restaurant model. This provides the search capabilities we are looking for: with a simple query, users can easily obtain a list of Mexican restaurants. A disadvantage to this approach is that doesn’t offer flexibility to restaurants who don’t fit neatly into one cuisine or another. For restaurants serving multiple cuisines, the flexibility of having a string variable `type_of_food` is superior. We debated between which approach to implement; but, then we realized that we could implement both. Our Restaurant model contains a foreign key which references a type of cuisine; but, the model also contains a string attribute that enables users to enter a description of the restaurant. 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 a restaurant's hours of operation in our Restaurant model. 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 repetitive and unnecessary. We might 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 in Restaurant Hours can read "Mon-Friday 8am-8pm" and another row can read "Monday 8am-8pm, Tuesday 10am-10pm." This method is flexible and enables storing less information in our database since many restaurants share similar hours (like 8am– 6pm). But this approach complicates our data since it makes Restaurants dependent on many relationships. We now choose to represent a restaurant's hours of operation by having seven strings representing that store hours for each day of the week (`mon_hours`, `tue_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 the Customer model. A Customer has a one-to-many relationship from itself to DishReview and RestaurantReview. Originally, we had a table with four attributes: `RestaurantReview`, `RestaurantReviewComment`, `DishReview`, and `DishReviewComment`. This table had a one-to-many relationship with our Customer, Restaurant and Dish tables. A customer could make a review of a dish and a restaurant at the same time. This method keeps a customer's reviews concise and in one place, but requires 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: `star_rating`, `dollar_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 while the previous method had a customer's reviews contain information about both a restaurant and a dish. Finally, we 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 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 restaurant or dish. A restaurant or dish page displays its average rating. 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 after each customer review and then divide the sum by the number of ratings. However, we don't want to traverse either of our review tables 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. Another approach is to keep track of the current sum of all the ratings for a restaurant or dish and its current number of ratings. If a customer leaves a rating for a restaurant/dish, we can quickly add his or her rating to the sum of the ratings and increment a counter that keeps track of the number of ratings for that restaurant or dish. The average rating is then quickly recalculated using the updated information (the sum of the ratings divided by the count of the ratings). However, this approach introduces a possibly inconsistent database. Calculating the average rating using this method 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 or 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 programming errors.

Our original Restaurant model included a cost field. This field was removed mainly due to legal reasons. We cannot advertise a price for a restaurant unless it's generated by our reviewers. When a reviewer rates a dish they give it a star rating and a dollar rating to indicate the dish's cost. For example, if a user has the best burger in the world and feels it didn't cost him or her as much as it should have then he or she would give the burger a five star rating and a one dollar rating. Similarly, if a user pays for an expensive steak but didn't enjoy it at all because the steak wasn't cooked properly, he or she can give it a one star rating but a five dollar rating.

Every time a user leaves a star rating, the dish or restaurant's `num_star_ratings` variable is incremented and the `sum_star_ratings` is also adjusted accordingly. The dish or restaurant's `avg_star_ratings` is computed from these two values. A similar process occurs when a user leaves a dollar rating for a particular dish. When the average star rating or dollar rating for a dish is displayed, it is the result of a query to our database. Furthermore, to display the average cost of a dish at a particular restaurant, the cost can be computed using this information in our database. In future implementations, we will enable a method for inputting and displaying star ratings for dishes and restaurants.

Our site is also depends heavily on images. Each restaurant, reviewer, and dish has an image attribute. We had three options to make retrieval and placement of these images dynamic. First, we could store a web address in the database of every image we need that is an image of a restaurant or dish, or a Facebook profile picture of a reviewer. The downside to this approach is that links become invalid or change. Furthermore, servers can be slow increasing the load time from various sources. Secondly, we could store the image in our database as a BLOB object. However, this slows down the performance of the database drastically as more images and items are added. Thirdly, a folder on our server which has three subfolders for dishes, restaurants, and users can store the images. This is our chosen option because it is faster and still enables us to identify images uniquely and dynamically. In these respective folders we store the static images with their filenames matching the item's ID in our database. Now, a page loads an image from our server in a dynamic manner because it depends on the item's ID.

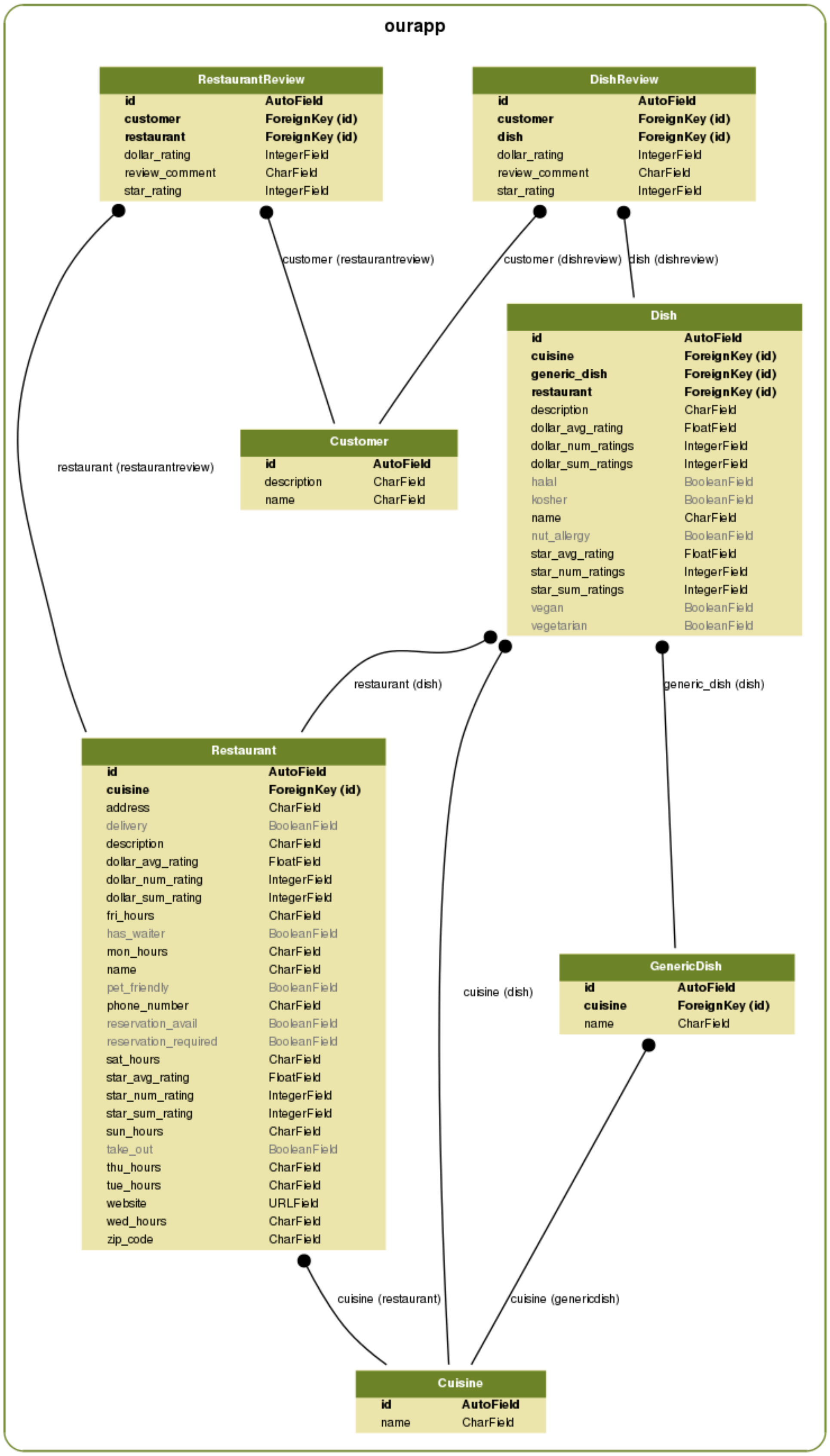
Templates and Refactoring

At the end of phase one, we were up to a total of 34 different HTML files. As part of our final review, we inspect everything for dead links, diagnose Javascript bugs and fix any other number of problems in dozens of files. This situation is less than ideal, as even a minor change to an item included on multiple pages required the modification of dozens of files. Fortunately, Django enables us to reduce redundancy in our code and improve efficiency through the use of its template language. The previously created static HTML pages are methodically broken down into modular components, and those components are likewise broken down further into subcomponents. The following structure is used: master.html is the primary template, and contains Django directives (specifically, the block tags) to allow subtemplates to define properties like `TITLE`, `HEADER`, `MAIN_CONTENT` and other content dynamically; subtemplates including *restaurant*, *splash* and *dish* contain block tags that replaced by dynamic content, and these templates extend *master.html*.

Each of these sub-templates also make use of templates like *review_list.html* and *menu_list.html* to handle iteration over the requested data. For example, *menu_list.html* generates a formatted unordered list from the `dish_results` functon defined in `views.py`; in other words, the for loop will create the menu for the restaurant in question. The template *review_list.html* does the same for reviews of a dish or restaurant. Although the aforementioned templates may seem to comprise many layers of abstraction, their use allows us to effectively separate content from presentation. Whereas we would have previously needed to edit multiple files just to add or remove a menu item from the top navigation bar, we can now edit *menu.html* to have the same effect. Javascript and CSS had previously been embedded within each page; but, these too are now in external files included via Django's `{% static %}` directive.

Logical Data Model

This is a UML class diagram of our Django models. This feature is available in the Django extension graph models. It created a **GraphViz** dot file based on our app's `models.py` file. A Cuisine can have one to many Restaurants, Dishes, and Generic Dishes. A Restaurant can have one to many Restaurant Reviews and Dishes. A Customer can have one to many Restaurant Reviews and Dish Reviews. A Dish can have one to many Dish Reviews. A Generic Dish can have one to many Dishes. In the next implementation of Kravester, the descriptor Generic Dish will be changed to Food Category. These relations are presented by a class UML diagram on the following page:



RestaurantReview	
id	**AutoField**
customer	**ForeignKey (id)**
restaurant	**ForeignKey (id)**
dollar_rating	IntegerField
review_comment	CharField
star_rating	IntegerField

DishReview	
id	**AutoField**
customer	**ForeignKey (id)**
dish	**ForeignKey (id)**
dollar_rating	IntegerField
review_comment	CharField
star_rating	IntegerField

Dish	
id	**AutoField**
cuisine	**ForeignKey (id)**
generic_dish	**ForeignKey (id)**
restaurant	**ForeignKey (id)**
description	CharField
dollar_avg_rating	FloatField
dollar_num_ratings	IntegerField
dollar_sum_ratings	IntegerField
halal	BooleanField
kosher	BooleanField
name	CharField
nut_allergy	BooleanField
star_avg_rating	FloatField
star_num_ratings	IntegerField
star_sum_ratings	IntegerField
vegan	BooleanField
vegetarian	BooleanField

Customer	
id	**AutoField**
description	CharField
name	CharField

Restaurant	
id	**AutoField**
cuisine	**ForeignKey (id)**
address	CharField
delivery	BooleanField
description	CharField
dollar_avg_rating	FloatField
dollar_num_rating	IntegerField
dollar_sum_rating	IntegerField
fri_hours	CharField
has_waiter	BooleanField
mon_hours	CharField
name	CharField
pet_friendly	BooleanField
phone_number	CharField
reservation_avail	BooleanField
reservation_required	BooleanField
sat_hours	CharField
star_avg_rating	FloatField
star_num_rating	IntegerField
star_sum_rating	IntegerField
sun_hours	CharField
take_out	BooleanField
thu_hours	CharField
tue_hours	CharField
website	URLField
wed_hours	CharField
zip_code	CharField

GenericDish	
id	**AutoField**
cuisine	**ForeignKey (id)**
name	CharField

Cuisine	
id	**AutoField**
name	CharField

Tests

Unit Tests

For phase two of the unit testing we hard code three instances of each table we have in our database: three dish reviews, three restaurant reviews, and three restaurants. Django loads this data into a database used specifically for testing. In `urls.py`, we have names for the URL mappings that correspond to our RESTful API. Our test suite calls these URL patterns with the appropriate parameters for JSON returns, which we then compare with the data in Django's temporary testing database. We debated two different styles for coming up with the accepted value to compare with the JSON responses. First, since our database store static data, we can hardcore the expected value as a python dictionary. Alternatively, we can inspect what is in the testing database and dynamically generate the expected results for an API call. At first, it seems that it doesn't matter how the accepted "true" value to be compared with the JSON response is generated, so long as it is correct. However, hard coding the expected values makes our test code very fragile. If we want to add or remove anything from our test database, we have to back and change all hard coded values in our test suite in order for our tests to continue pass after making changes.

We dynamically generate the expected values of our API requests. One of the initial implementations to generate these values was to use the Django REST Framework serializer to convert the instances of the objects in our database into JSON dictionaries. But our API uses this exact implementation to generate JSON responses, so the initial implementation is a bit circular: using the same mechanism to generate expected values and return values. Our final implementation uses a method in each of our models, `getDict()`, to convert an instance of a model into a dictionary that is formatted in the same way it would appear in a JSON response. This implementation is challenging because each testing function calls the function `model_instances()` that returns a dictionary of the instances of objects in the testing database. Since this function is repeatedly called during testing, new items are continually added to the testing database. Because of this, the primary keys (ID numbers) used to make our API calls continuously change during testing, since their ID numbers are incremented or changed after every call to `model_instances()`. Our way of circumventing this problem is by using the `objects.all()` method for each model to return a query set of the objects in a table at the beginning of each function call. Since we know the amount of instances that we created (3), we know exactly how many instances to iterate through for our tests. This method enables us to do the comparisons accordingly.