Phase Three 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, "What do you want to eat?", why is it that most people respond with the name of a restaurant or a cuisine instead of an 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 this page contains 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 him or her. This page also has a list of every restaurant the reviewer has reviewed/rated and 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 functionality 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 is 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 times we have in our database for restaurants. A future feature of our search functionality, similar to websites like *yelp.com*, will be to enable users 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. In this implementation, we simply enable users to search Kravester with contextualized **AND** and **OR** results.

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 will 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.

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 dish is 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 as restaurant location or only vegetarian options.

API Design

Introduction

The first step in designing our RESTful 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.

Using Serializers to respond to GET Requests

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.

The Kravester API Hierarchy

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

Introduction

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.

Restaurant and Customer: One-To-Many Relationships

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.

Using Models for Restaurant and Dish Ratings

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.

Using Models for Restaurant, Reviewer, and Dish Images

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

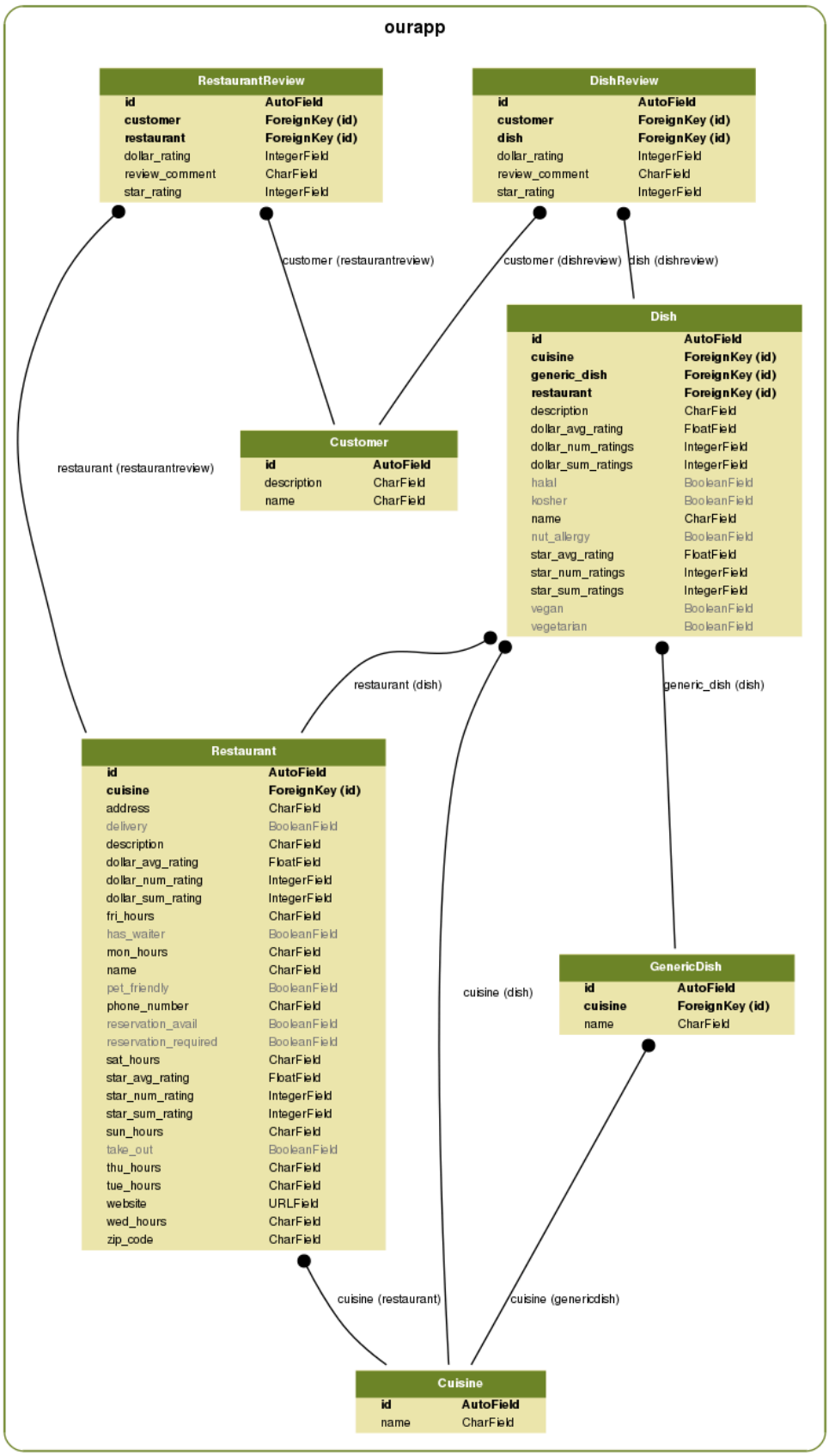
At the end of phase one, we were up to a total of 34 different HTML files. As part of our final review, we inspect everything for dead links, diagnose Javascript bugs and fix any other number of problems in dozens of files. This situation is less than ideal, as even a minor change to an item included 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:


ourapp
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
customer (restaurantreview)
customer (dishreview)
dish (dishreview)
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 (restaurantreview)
restaurant (dish)
generic_dish (dish)
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 (dish)
cuisine (restaurant)
cuisine (genericdish)
Cuisine
id AutoField
name CharField

Searching Kravester

Introduction

To understand our search implementation, one needs to know how our requirements and techniques evolved over time. Originally, we planned to do a single, one size fits all text search with no separate searches for restaurants, dishes, or anything else. Since many people use **Haystack** we looked into using it ourselves. Haystack's documentation states that it searches through Django models. Implementing Haystack on **PythonAnywhere** is too complicated; therefore, we attempted to implemented model searching ourselves. This process proved to be a nightmare. In this implementation, all searches need to be handled on a case by case basis. For instance, finding a search term match inside a dish's name is different from finding a search term match in a restaurant's name or in a dish's description. The problem is as follows: if a search term mach is found in a dish's name, not only do a URL and search result string need to be generated, but also the URLS and search result strings for any other related dish. Finding a search term mach in a restaurant's name is also a different case. A URL and a search result string for the restaurant's cuisine type also needs to be generated in this case. This leads to a very fragile system. In the future, if our dish template doesn't include similar dishes, this search feature will break.

Parsing HTML to Implement a Search Feature

Our solution to implementing a search for Kravester does an actual text search through the HTML of Kravester's pages. This enables us to not have to deal with things on a case by case basis. It doesn't matter whether or not a search term match is found in a

dishes' name, a restaurant's name, a dishes' description, or anywhere else. All that is required is to construct a list of URLs containing every URL that we want included in our search index and then every URL can be processed by performing a text search through its contents. Constructing the list of URLs is easy to do: loop through all of the restaurants in our database and insert their IDs into the URL template for restaurants. The same thing is done to construct URLs corresponding to dishes. We can also manually add URLs for single pages such as the splash page or the about page. After obtaining the list of URLs that we want to include in our search index, we need a way to get the contents of their rendered web pages. A simple solution uses Python's requests module to grab the HTTP responses that PythonAnywhere sends back to a requester. It is out of the scope of this document to explain why, but this leads to deadlock. Instead, we obtain the rendered HTML for our pages internally without needing to make any HTTP requests. This also has the benefit of being faster than waiting for HTTP requests and HTTP responses.

Using Beautiful Soup to Find Page Elements

After obtaining the rendered HTML for our pages in the form of strings, we still have to search through them and to return contextualized search result strings. We need a way to strip out all of the HTML and JavaScript from the rendered HTML strings. We use **Beautiful Soup** to accomplish this. More specifically, we use Beautiful Soup's "find_all" feature which returns an iterable of strings where each element corresponds to the text between a different HTML `<p>` tag and/or any other tag you desire. We also found a way to strip out all of the HTML with Python's `htmlparser` module, but this has the problem of not stripping JavaScript code. Once we have the actual text extracted from our pages (without any HTML/JavaScript), the test is searched for a match. First,

we need to split our search query string up into a list of search terms. As a simple, initial solution we took the query string and called `split()` on it to convert the string to a list of words. We first replaced punctuation in the query string with whitespace (using string's translate method) and then converted the whole search query to lower case before calling `split()`.

Determining AND and OR results

Now that we had our list of search terms and we also had a way to extract the text from pages in our website, the next step is to determine if a page's text contains a search query match for **AND** searches and for **OR** searches. For an **AND** search this is easy to determine: check that every search term in a page's text using code like the following:

```
text = text.lower()

#Return None if any search term wasn't matched.
for term in terms:
    if not term in text:
        return None
```

For an **OR** search this also easy: check if a page's text contains any search terms or not. We achieve this using code like the following:

```
for term in terms:
    if term in text:
        foundSomething = True
        break

if not foundSomething:
    return None
```

There surely are slightly more efficient solutions than these snippets of code, but these snippets have the benefit of being simple and they are also fast enough for our purposes.

Now that we have determined whether or not a particular page contains a search query match, if it does not contain a match then a search result string needs to display to the user. There is a lot of room for creativity and complexity here. Since we don't want our search result strings to get too long, we need a way to decide exactly what to display using a limited number of characters. Since we extract the text out of our web pages one paragraph at a time, it makes sense to artificially insert ellipses at the end of each paragraph so that the user knows where one paragraph ends and where another begins. We also had to decide exactly which paragraphs to include and how to truncate them. As appropriate for the scope of our project, we opt for a simple solution. We include characters up to one hundred characters before and one hundred characters after the first search term match in each paragraph for a particular page while separating separate paragraphs with ellipses.

Formatting Search Results

After concatenating these results from each paragraph, we then truncate this final resulting string to five hundred characters and apply ellipses if necessary depending on whether or not the final truncation cuts into the middle of a paragraph or whether it cuts off exactly on the boundary between two paragraphs already (and thus there would already be ellipses at the end of the resulting string). We also handle the case where truncation cuts into the middle of ellipses. In general, this works very well because our paragraphs are rarely very long; so, everything gets included and no truncation is necessary. It is important to note that we decoupled the code for formatting the search results from all of the other code so that it is easy to change the way we are displaying search results once we have found a web page that matches a query. The signature of this

function is `formatResults(pList, terms)`. Given a list of paragraphs from a particular page that all contain search term matches and given a list of search terms, it generates a search result string.

After doing all truncation we are nearly done constructing our search result string. The last thing to do is to go back over the search result string and place HTML strong tags around any word that is a search term in order to fulfill the contextualized search result requirement. It is important to add these tags after truncation so that we don't accidentally truncate a closing tag out of the text while leaving an open tag in place.

Separating Search Results by Restaurant and Dish

After creating all of this framework, we decided to completely change the dynamic of our search feature. Instead of being a plain text search, we wanted a restaurant search and a dish search instead. If doing a dish search for “taco”, we only want a search result for “Blackened Fish Taco”s URL, and not a search result for “Chicken Quesadilla”s URL even though the blackened fish taco is listed as a related dish on the chicken quesadilla page. Under these new conditions, we can implement searching by our initial solution through model searching; but, since we already have our other search framework working perfectly, we adapt that framework to our new purposes. Instead of having Beautiful Soup extract text between any paragraph tags (where it could find information not pertaining to a particular restaurant or dish) we have it only extract text between `<h1>` tags and `<p>` tags that have the HTML class "description" attached to them. This is a simple way to search through a dish/restaurant name or description without including any other information from the dish or restaurant's page. We also find our search results to be quite responsive temporally.

Conclusion

Once we have a list of search results, we need an interface to return the data to our front end developers. Together with our front end developers, we set up a URL, `/search/<query>` that our front end developers can go to in order to obtain search results. They fill in the query section of the URL, and a JSON HTTP response is returned to them with the search results. We opted to return a list of dictionaries with the following keys: `url`, `text`, `type`, `modelType`, `id`, and `name`. The `url` key maps t the URL containing the search result match. The `text` key maps to the search result string to be displayed to the user. The `type` key determines whether the search result is an **AND** or an **OR** search result. The `modelType` key is used to determine whether the search result is found in a dish or a restaurant. The `id` key maps to the dish/restaurant's ID and can be used to obtain a picture of the dish/restaurant. The `name` key maps to the name of the dish/restaurant in order to give further context about a search result.

Unit Tests

Using Django's Temporary Testing Database

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. 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. We then use Django's

test client to make calls to our API. These calls look something like this :

```
response = self.client.get(reverse('adishjson',
     args=[Dish.objects.all()[0].id, 'json']))
```

Note that we use `reverse()` in order to generate our API URLs instead of hard coding them. This will make it easier if we decide that we want to change the URL interface to our API. Don't worry about the args portion of this line of code for now, it will be explained soon. After obtaining the response for an API call, we now need a way to verify that this response is correct. We need to compare the response we obtain via the API call to a known true, accepted value. 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. Then verifying the correctness of a JSON response would look something like this:

```
self.assertEqual(json.loads(response), {'Name' :
'Alfredo pasta', 'id' : 3, ...});
```

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 change all hard coded values in our test suite in order for our tests to continue pass after making those changes. For instance, if we were to change our API to return `Name` as `Dish Name` instead we would need to update the previously illustrated line of test code as well. Or, if we were to add another model instance to the test database, the primary keys of all of our other model

instances could change.

Testing API Responses using Built-In Dictionary Methods

We dynamically generate the expected values of our API requests. Now verifying the correctness of a JSON response looks something like this instead:

```
self.assertEqual(json.loads(response),
dishObject.getDict());
```

Under this new scheme, if we decided we want to change our API to return `Name` as `Dish Name` instead, we would not need to change several lines of test code similar to the previously demonstrated line of code. We would only need to change our dishes' `getDict()` method and this change would propagate to all tests that call the `getDict()` method. 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. Previously we had mentioned that we would later explain the `args` portion of this line of code:

```
response = self.client.get(reverse('agdishjson',
args=[Dish.objects.all()[0].id, 'json']))
```

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.

Search Tests

Introduction

Testing our search feature was a bit trickier than testing our API calls. This is because there is no defined correct output for search results. For our API calls we were able to simply check whether or not the JSON results of a particular API call exactly matched the expected results. There are no exact correct results for searching, however, because which results are returned and how the search result strings are formatted isn't well defined. Many aspects of our search results are well defined; therefore, we were still able to analyze many features of returned search results.

Testing Search with HTTP Requests

After creating some model instances in our test database, the first thing we need to do in each search result test is to obtain search results using the interface we documented in the search portion of this tech report. We do this using a snippet of code like the following:

```
response = self.client.get(reverse('search',
           args=['bacon burger']))
searchResultsStr = str(response.content)[2:-1]
searchResults = json.loads(searchResults)
```

This snippet of code uses Django's test client to make an HTTP request to our search URL. The HTTP response contains the search results in JSON form and is converted with `json.loads(...)` into a Python list of dictionaries. Note that we use the reverse function from `django.core.urlresolvers` so that we don't have to hard code the search URL. This makes our system easier to update in that if we want to change the URL aspect of our search interface, we can now do so without having to update our tests.

Once we have the search results in `searchResults` (a local variable), we inspect the results to make sure they are valid. We analyze the search results in a variety of ways. For instance given a model's ID returned in a search result, we verify that the name of the model, and the URL for the model that are included in the search result match the correct value associated with that model ID (As in the above snippet of code, we generate the expected URL using the reverse function to avoid hard coding anything). Even though the exact search result string isn't well defined and is subject to a variety of formatting strategies which could be changed in the future, we were still able to test

certain aspects of the search result string for validity. For instance we know that every search result string should contain at least one matching, contextualized word. Thus we check to make sure that the matched word is found in the search result string with HTML strong tags around it (since our search interface adds strong tags around matched words in order to provide contextualized search results).

Working with Tags

We also make sure that opening strong tags appear before closing strong tags, and that strong tags are not nested. This can be accomplished fairly easily with the following code:

```
pat = re.compile("(<strong>|</strong>)")
matches = re.finditer(pat, string)
while True:
    try:
        match = next(matches)
        self.assertEqual("<strong>", match.group())
    except StopIteration:
        break
    try:
        match = next(matches)
              self.assertEqual("</strong>",
              match.group())
    except StopIteration:
        self.assertTrue(False)
```

`self` refers to an instance of a unit test class, thus we are able to make calls such as `self.assertTrue(...)`. Matches will be an iterable of match objects corresponding to instances of either `<strong>` or `</strong>` inside of the search result string. We start by making sure that the first match corresponds to `<strong>` and then we make sure that there exists a subsequent match corresponding to `</strong>`. Notice that if there are no more opening strong tags which triggers `StopIteration` then we just break out of the loop. However, if the expected closing strong tag is not found, then when `StopIteration` is triggered, we use call `self.assertTrue(False)` to signal that the test case failed.

Conclusion

In general, the number of search results and which search results are returned is undefined. If a search query yields thousands of results, we wouldn't want to return all of them. However, it is reasonable to assume that if a search query only yields one or two results, then they should be included in the search result list for that query. Under this assumption we include tests that we know will only yield one or two search results, and we verify that the number of search results equals the expected value (by simple checking the length of the returned list of search results). We also test the special case where a search query yields no results.

Implementing The Austinites' ACL API

Introduction

The Austinites' website has an attractive landing page. The page's pastel color scheme and minimal theme invites users to explore the website driven by their API without initially feeling overwhelmed and the rich detail they provide for the resources they cover. The Austinites maintain an Austin City Limits (ACL) music festival API. The API is read-only and enables users to retrieve information related to the festival's artists, stages, and sponsors. On the Austinites' website, each category's information is organized by year. For example, a user scanning the artists page sees that Asleep at the Wheel has played ACL for the past three years since they are listed under each year of the page. Similarly, a user scanning the sponsors page sees that BMI(an entertainment company) has sponsored ACL for the past three years and Samsung Galaxy has sponsored for the last two years.

Using the ACL API in a Meaningful Way

So how can the information in the ACL API be used in a meaningful way? There are two main audiences for music festivals in general: those attending to enjoy the festival and those attending because they are involved in the music industry. Our implementation of the ACL API targets the audience that is attending the festival for entertainment. The nature of large music festivals is that there are about a dozen bands with large followings (whose fans purchases a large share of tickets) and three times as many bands with small to moderate followings who play the festival to get exposure to a larger audience and often play during the afternoon and early evening. The ACL API

contains general information about artists like genres and record labels and media information like YouTube channels, biographies, and Facebook pages. Our implementation of the ACL API enables users to discover Austin City Limits artists by their location, rather than by music label, genre, or year. It's aim is to enable users to look at the artists they discover in a different way. Is there a particular sound that bands from Los Angeles or Atlanta have? With enough information, users might begin to get a feel for the answer to that question. Using geography as the distinguishing element among artists provides a different user navigation experience; it suggests a possible alternative to classifying artists regionally by noting any similarities among them.

Creating a Google Map Using the Google Geocoding API

The first step in building the implementation is to create a map, and put a marker at a location for each artist in the ACL API. The **Google Maps Javascript API** enables us to create a map variable in our script and define it's attributes. This implementation was fairly straightforward. Since the ACL API has an `origin` attribute for each artist, which is just the name of a city, the Django template file contains a Javascript loop that creates a list of origins(city names) based on each artist's `origin` attribute. The **Google Maps Geocoding API** is used to find the latitude(`lat`) and longitude(`lng`) of each city in the locations list, and a marker is then created for that city. Additionally, a function is defined such that when each marker is clicked, two things happen. First, the HTML in the `outer-map div` is written dynamically to list the names of the artists from that location, their genre/record label and the number of years each artist has played the ACL festival. Second, the HTML for each artist's modal is written dynamically. The modal is a separate window which appears when a user clicks on the name of the artist that

contains relevant information about that artist. One of the difficulties with this implementation is that the Geocoding API limits the number of queries that an API user can make per second (or millisecond). Since the content is generated dynamically from the ACL API, it makes no sense to hardcode latitude and longitude locations of popular cities and hope that all artists are from there. Javascript has a function `setTimeout` which takes two parameters: a function and its parameters and a time delay in milliseconds. The original solution looped through the `locations` list, and called the function `geocodeOrigin` to get the correct latitude and longitude for creating a marker. However, looping through the locations list ignored the case in which the geocoding response returned a query limit error.

The `geocodeOrigin` function originally only needed a location parameter to geocode. A geocoding request executes a callback function upon getting a response where the results of that response exist only within the callback function's body. If looping through one location in the list and passing that location to `geocodeOrigin` returns an *OVER QUERY LIMIT* error, then that location is skipped in the process of creating markers for the map. An online Google maps tutorial[1] helped with issue by showing that a function can also be passed to `geocodeOrigin` that handles the logic of the original for loop and get's called within the body of the geocoding response's callback function if there is a query error. Additionally, two global variables keep track of the current index in the locations list (`nextLocation`) and the delay time(`delay`). This way, if a query limit error is returned, we increase the delay of `geocodeOrigin`'s execution, and make sure to not increase the current index in the

[1] http://econym.org.uk/gmap/geomulti.html

locations list. Both of these variables are accessible within the callback function's body because they're global. Then the function handling the logic of iterating through the locations list is called with the body of the geocode request's callback function. In some sense, this enables the `geocodeOrigin` function to call itself until the request no longer returns a query limit error response.

Using Navigation to Drive Dynamic Content

Each marker created as a result of geocoding request drives the dynamic content of our API page. Clicking a marker calls several functions which are responsible for dynamically creating new HTML code for both a modal window(from Twitter Bootstrap) and the `outer-map div`. The modal window is designed to enable users to identify whether or not they like an artist by providing relevant information, like a YouTube video which plays a song by the artist and lists similar artists that users may have heard of. The modal window is a good choice for displaying information because it prevents users from having to navigate away from the API page. Instead, a user can simply close the tab after being given a brief introduction to an artist and can continue to explore the original list of artists from a location. While the ACL API did provide a YouTube video for each artist, our implementation also uses the JQuery method `getJSON` to interact with the **YouTube API** and request a JSONP response containing the most relevant video related to an artist. Different parameters can be passed for each request, but our implementation only makes sure that the length of the video is between 4 and 20 minutes. This method is used as a backup in case the ACL API does not provide a YouTube video value for an artist. The YouTube video is loaded into an `iframe` within the modal's body that is initially written by Javascript code with no source value. When users click on

the name of the artist to bring the modal window into view, the source value for the iframe is set. Otherwise, the YouTube video would play in the "background" when the modal window isn't displayed. Similarly, when users close a modal window, the source value for the `iframe` is removed. These two processes were implemented by giving the `iframe` a unique ID element and using JQuery to set and remove the `iframe` source value.

The artist modal window has both an image and a video for an artist and also contains biographical information about the artist along with similar artists and frequent descriptions of the artist. Since the ACL API does not store information about similar artists or frequent descriptions of artists, we used the **Echonest API** along with JQuery's `getJSON` method to retrieve that information. Echonest stores a lot of information related to artist, genres, and is even used to create dynamic playlists that respond to user input. By querying within the artists category of the Echonest API for `similar` and `terms`, we can iterate through the results and use however many we'd like to attribute to an artist. The results for an artist's descriptions often exceed 100, and each description is given a weight between 0 and 1 to determine how often the description is used for that artist. Additionally, resulting descriptions are returned in order from most frequent to least frequent; so, we use only the first several results of each response to build a list of common descriptions for each artist. All of these results are generated dynamically by getting the name of the artist from the ACL API and passing it as a parameter in the Echonest API request.

Conclusion

The **Flickr API** is also used on the page to request a list of photos from a photoset of live pictures from ACL's Flickr account[2]. A random set is chosen every time a user visits the page by generating a random integer for the index of a photo. This feature was included to make the page more visually appealing. The result is a page that looks attractive and simple, yet contains a large amount of information. Just by clicking on markers on the Google map, users can explore artists from different areas of the world joined together at ACL. User can find out how the artists are commonly described and instantly see one of their music videos. The ACL API also includes artists' Facebook information; so, if users like an artist they discover, they can use a Facebook button in the modal window to "like" that artist and began to get updates about that artist in their Facebook news feeds.

[2] https://www.flickr.com/photos/aclfestival/sets/72157636188215713/