Regional Foods IDB

Witty CS Pun

James Bradford

Calvin Aisenbrey

Sammi Cooper

Binwei Lin

Jishen Li

Anthony Lopez

Introduction

The Problem

Regional Foods IDB is a database-backed website similar in function to the familiar IMDB. RFIDB documents related recipes, their regions of origin, and the famous chefs who cook them. This IDB is useful for anyone who needs a specific regional recipe, wants to follow his favorite T.V. chef, or is interested in a certain region's typical dish. As this is the second iteration of the project, the website is more exhaustive in terms of available recipes, regions, and chefs than the previous iteration. Because the web pages are dynamic, the website facilitates the introduction of new data. The project required that we design a RESTful API, a set of django models corresponding to that API, and ten dynamic HTML pages for each of our models.

Use Cases

As mentioned before, this website provides much desired utility to a variety of visitors. For example:

- I'm the mood for something Southern…

  This user may explore regions after having navigated from the simple and no-nonsense homepage by following the link to that hub page. A quick glance at our visually clean and uncluttered grid layout will reveal a set of options. The user would intuitively click on the picture of the Southern USA and open its page with a list of southern chefs and a list of southern recipes.

- I just saw Gordon Ramsay on the Youtubes. I want to cook like that guy!

  Aspiring chefs can go to the website, then click on chefs and they will find Gordon Ramsay, among a plethora of other famous chefs, on the page. When they click on the link, there will be a brief introduction of Gordon Ramsay; also, they can follow him on Twitter on the right of the website page. Then the users will be able to find Gordon Ramsay's recipes on the right column. Just with a simple click, users will be able to see ingredients of that food and every detailed step of how to cook the same dish as Gordon Ramsay.

- I eat the same thing every day...I want to try something new! Anything!

  Visitors need look no further than our recipes page! Well, maybe a little further. Just as the previous user simply followed the appropriate link from home, users will do the same in this instance to find all of our recipes all in one place. These recipes come from all over the world and many are backed by top celebrity chefs. Again, all steps and major details are included on our recipe pages.

Design

RESTful API

Implementation

A RESTful API is an API designed to adhere to the principles of REST (Representational State Transfer). The API should clearly define the resources

(represented by the web pages) of the website and provide GET, PUT, POST, and DELETE functions to manipulate the resources. Our API only provides GET so that no others may delete or add to our website, i.e. our data is read-only. Instead of providing PUT, POST, or DELETE, we manipulate the resources ourselves inside the database. The GET functions we have defined in our API are implicitly called when a URL is specified and provide the information specified in the body of each GET response (called a payload) in JSON format.

The API provides a way use GET to list all objects of each page type (i.e. region, chef, recipe as “api/<page-type>”) and their data as a 200 response formatted in JSON. The API also provides a way to get any object by its primary key, which is automatically created by Django, (as “api/<page-type>/<primary-key-for-object>”). When GET is called on a valid object, a 200 response is returned, meaning that JSON is returned as the response body. This JSON contains keys that correspond to the fields of each object (as defined by the Django models), and values that contain the values of each corresponding field that belongs to the requested object in our database. The dictionary that has an ID (i.e. primary key) that matches the primary key specified in the GET call is returned by the call. We also specify a 404 response, which will be called when an attempt is made to GET a page that does not exist in our database or in our API.

Interface

Http Verb	**Input: root_url/api + resource**	**Action**

GET	/chef/	return .json of all chefs and their data
GET	/chef/{pk}	return .json of all data of chef associated with {pk} primary key
GET	/region/	return .json of all regions and their data
GET	/region/{pk}	return .json of all data of region associated with {pk} primary key
GET	/recipe/	return .json of all recipes and their data
GET	/recipe/{pk}	return .json of all data of recipe associated with {pk} primary key

Example GET on chefs

Typing http://regionalfoods.pythonanywhere.com/api/chef/1 in the URL returns:

[{ "pk": 1,

"model": "library.chef",

"fields": {

"bio": "Wolfgang Johannes Puck (born Wolfgang Johannes Topfschnig; July 8, 1949) is an Austrian-born American \r\n celebrity chef, restaurateur, and occasional actor. Wolfgang Puck restaurants, catering services, cookbooks \r\n and licensed products are run by Wolfgang Puck Companies, with three divisions.",

"birth_date": "1949-07-08",

"name": "Wolfgang Puck",

"youtube": "http://www.youtube.com/embed/Gc6B_N4eRoA",

"region": 1,

"style": "California, French, and Fusion",

"twitter_link": "https://twitter.com/WolfgangBuzz",

"image": "/static/chefs/wolfgang_puck.jpg",

"birth_place": "Sankt Veit an der Glan, Austria",

"twitter_id": "492022886987603968"

}

}]

Django Models

The Django models define the data that we are presenting with our website. The models are Python classes that contain various fields to define and describe each type of page we are using and how these pages will behave. These classes' fields also define the relationships between our page types. The classes are used to create objects that populate the data tables of our database. We have three unique models that correspond to the groups of pages on our website: Region, Chef, and Recipe. The fields for each object contain many different types of attributes that we will use in various ways. For example, some fields are simply the name of the object and will be listed along with other short fields in our web pages, and others may be used to embed media in our web pages, such as Youtube videos, Twitter feeds, and Google maps. Each model class also has a "__str__"

method that returns the name field of the model, so that when “str()” is called on an object, the call returns the name of the region, chef, or recipe.

Region

The Region model contains a character field for the region’s name, an image field that contains a url for an image corresponding to the region, a text field for a brief and general description of the region’s food, and a text field that contains information for an embedded Google map of the region.

Chef

The Chef model contains a character field for the chef’s name, an image field that contains a url for an image of the chef, a foreign key for the region of the world to which his cooking relates, a date field for his place of birth, a date field for his date of birth, a text field that contains the cooking style of a chef, a text field that contains information for an embedded Youtube video, a text field that contains a link to the chef’s Twitter page, a text field that contains a widget ID which allows us to embed each chef’s twitter feed into a web page, and a text field that contains a brief biography for the chef.

Recipe

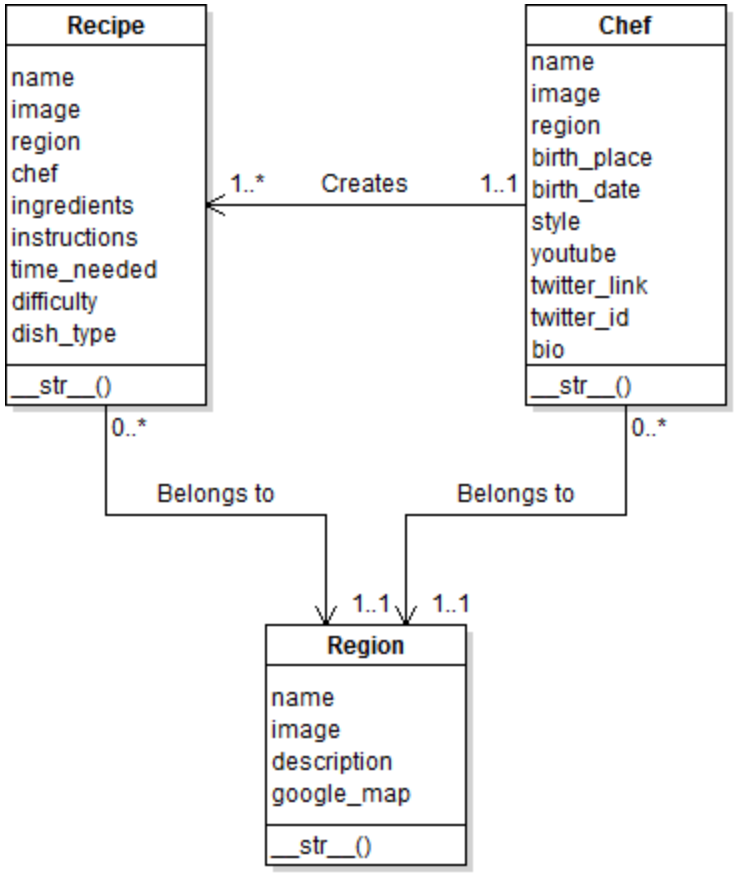
The Recipe model contains a character field for the name of the recipe, an image field that contains a url for an image of the final product of the recipe, a foreign key for the region to which the recipe belongs, a foreign key for the chef who created this recipe, a text

field that contains the ingredients for the recipe, a text field that contains the instructions for the recipe, a character field that contains the amount of time needed for this recipe (in hours and minutes), a character field for the difficulty of the recipe (which is represented as a choice list where ‘E’ corresponds to ‘Easy, ‘M’ corresponds to ‘Medium’, and ‘H’ corresponds to ‘Hard’), and a character field to represent the type of dish this recipe is (i.e. entree, side dish, or dessert).

Model Relationships

Many Chef objects may belong to one Region. Many Recipe objects may belong to one Chef. It follows that many Recipe objects may belong to one Region. These relationships are defined using foreign keys, which must be objects derived from other, related models. The fields that contain foreign keys will be used to link to other pages on our website. For example, a chef object’s page will link back to the page for the region object that is specified as a foreign key for the chef object in our model.

UML Diagram for Django Models



Dynamic Pages

Database

We created our MySQL database through Pythonanywhere. Using the command ‘python3 manage.py syncdb’, we were able to synchronize the Django models to the database. The synchronization converted the models to SQL tables, which we then manually populated through the ‘/admin’ page provided to us by Django. Because Django provides a wrapper for database manipulation and creation, the process was mostly automatic. We did not have to create a schema for our database or even write any SQL.

URLs

The ‘urls.py’ file specifies which URLs for our website are valid. In this python file, a urlpatterns object is created that contains patterns formed from the url() function (imported from django.config.urls). The url() function is passed a URL string (the URLs are interpreted with regular expressions to find the associated view) and a view, or a template transformed to act as a view. Through this function, a URL typed into a browser will call upon a view to find the content in the database specified by the URL.

Views

The ‘views.py’ file contains functions for each type of page that we want to generate dynamically (resource hubs and resources). These functions take in an HTTP request, and optionally a string that is taken from the URL and specifies which object’s page we’d like to generate. The functions for resource hubs do not have this optional string. The resource hub functions create an iterable containing every object of that resource type, put that list into a dictionary, and use the function render_to_response() (imported from django.shortcuts) to send that dictionary to a specific template page to be rendered. The functions for specific resources change the second argument given to the function (taken from the URL) to a name with all underscores replaced by spaces. They extract the object with that name from the database and place it into a dictionary. They also extract all items that have this object as a foreign key and place those into the dictionary. This dictionary is given to render_to_response() along with a specific template file with which it will render the information in the dictionary.

Dynamic Templates

We wrote dynamic template pages in order to build the foundation of the finished website. There are 3 sets of page types: resource hubs, resources, and static pages. The homepage remained static (and unchanged from the last iteration for the most part) because it was a single page of its kind. As for the about page -- we do not foresee the citation data being relevant enough to each resource to warrant associating it with each respective resource in the database. In addition, a great deal of effort was spent learning the fundamentals of rendering a web page with dynamic content.

Resource Hub Pages

Dynamic resource hub pages replaced the previous generations static hub pages. Instead of hardcoding data from and links to specific resource pages, we now dynamically access the database to build cells for each associated resource of that type of hub page. For instance, the chef hub page no longer just links to three chefs. In fact, it links to *all* chefs we have in the database. To access this data the html is given a context dictionary which contains a list of python objects of that particular hubs resource. Using template language in the html we can iterate through the objects and extract their data. Also, a Javascript function was needed to translate names so that they were in a format that we could use to link to other pages and correctly reference images (replaces spaces with underscores in names). The addition of new chefs to the database will automatically render a change in

the chef hub page. A wonderful side-effect of the grid layout of our design is the elegant and systematic evolution of our hub pages. New resources simply push into the next available slot of the grid and future-proof the growth of the hubs.

Resource Pages

The individual resource pages were replaced in the same manner as the hub pages. Each page receives a context dictionary with its respective resource and that resource's data is accessed similarly as in the hub pages. Growth on chef and region pages occur as more recipes are added to the database. This means that in their context dictionaries a list of recipe objects is also passed to the page. Each of these pages contains a side bar that provides a dynamic list of recipes that will grow longer and lengthen the containing column with additions. Also, regions similarly provide the expansion of its chefs list. Recipe pages do not really require growth. All of these pages utilize the same Javascript function as the hub pages to translate names.

Sochi Page (Other group's API demo)

Utilizing and embedding data from another website was as simple as writing a view (views.socchimain()) that calls that websites API GET method; in our case, we called requests.get() on 'http://ajhooper.pythonanywhere.com/api/country/?format=json' and 'http://ajhooper.pythonanywhere.com/api/athlete/?format=json' and subsequently converted the results to json format with .json(). From here manipulating the data devolved into the same process as what we had been doing for the rest of our pages: placing the data dictionaries into a context dictionary list and sending that context to be rendered with the correct dynamic page, olympicad.html. This html file matches any country and its

associated athletes with a region in our database if it exists. This work had to be done in the html in template language as, after many tries, we could not figure out how to do the matching the data in the view function. Once the data was converted to json in the view, it was trivial to manipulate the data and do comparisons against foreign keys in the template file. The idea was for this page to serve as an ad for the Sochi olympics and, thus, demonstrate how to display related data between websites.

HTML/Bootstrap

Throughout the process of building the dynamic web pages we had problems with formatting. Getting all of the divs, images, and items on the page to line up correctly with the objects sized correctly with the right amount of margins surrounding it proved to be a challenge. After getting ideas from past projects we decided to list our links to our three hub pages on our index.html page. At the top of every page on our website is a navbar which allows the user to easily navigate to each main piece of the site, and at the bottom is a floating bar which has our group name presented and serves our group signature. We decided that we would just list our three items in one row.

These three pages are the chefs, recipes, and regions pages. The chefs page would link to each chef, the regions page would link to each region, and the recipes page would link to each recipe. We decided that the format of the these pages should remain similar to our original index.html page. Therefore the items on each page would be shown with three items per row. This allowed for the pictures and titles to remain large and easy to read while not looking overly bulky.

After adjusting title lengths, cropping pictures, and setting up links to future pages we began to build the template page for chefs. We decided on a basic structure in which there would be a left column where important information about the chef would be and a smaller right column where their recipes, videos, and twitter feed would be. In this phase of the website we began to use Bootstrap's grid system extensively. Using this system helped format the page exactly how we wanted to, and also made it look visually appealing since Bootstrap adds useful CSS such as margins and borders.

For both regions and recipes we took a same approach at designing our templates. One large right column for important information about the recipe or region, and one small column on the right which has links to other relevant items on our website that correspond with the item being currently shown. On our recipe pages we included a picture of the food the recipe would make, the recipe to make it, time it takes to make, and the amount of servings it will produce. Additionally, the right column includes the region that the recipe originated from, and the famous chef who made the recipe. Our region pages display a picture of the region, and a Google Map of the region in the large left column. In the right column we have a description of the local cuisine along with recipes for food relevant for the region, and a chef from the region.

Search Functionality

Views Functions

To achieve a search functionality for our website, we copied and modified code from http://julienphalip.com/post/2825034077/adding-search-to-a-django-site-in-a-snap. This code contains three functions: normalize_query(), get_query(), and search(). We placed these three functions in our views.py file. Search() calls get_query(), which calls normalize_query(), which splits a given query string into an array of individual words, ignoring spaces and grouping together words bound by quotation marks. This array is then passed to get_query(), which takes in as parameters the normalized query string, a list of fields for a model, and a boolean that specifies whether the query should produce AND results or OR results. Get_query() then creates a complex query using Q objects which combine Django queries that ask if each term in the query string is in each field belonging to an object. The search() function then passes the result of get_query() to a filter on the database object two times for each model that we have (once for AND results and one for OR results), and then places the results into a context dictionary. The context dictionary contains the original query string that was typed into the search bar, and separate sets of entries for each type of result (six in total: AND results and OR results for each model type). This context dictionary is sent to the search_result.html template to be rendered.

User Interface

The search bar is provided by Bootstrap and appears on the navigation bar of every page. The template for the search_result page will appear when the user enters a query into the search bar and presses the “Search” button. When multiple words are given as a query, the page displays the two categories of search results (AND and OR) as panels with

hidden content that expand when clicked on. This was done using JavaScript and JQuery to determine the current state of the panel change its state as necessary. When only one word (or multiple words grouped in quotation marks) are given as a query, only the AND panel shows up, and it is automatically opened because OR results do not apply when there is only one word in a query. The content underneath these panels is grouped by model type and displays links to pages representing objects in our database that are relevant to the search query. Underneath these links, items in the search query are shown highlighted inside the field of the object that the items appeared in to provide context for the query. This was accomplished by finding which attribute for each result had the query inside of it and then parsed through the text in order to replace the text with a span tag which changed the color of the text. The simplest way to do this was to go through each word in the search query, find all the indices the word was at and an entry into a dictionary of the form “<index>”:<query_word>. At the end of the parsing function we sorted the keys so that they could be looped through. They needed to be in order so that when building the new string, which would have the span tags, we could get the plain text that did not match the search query. We did this by getting the substring starting at the index equal to the previous key plus the length of the word linked to the previous key and ending at the index equal to the current key.

Tests

Django Model Tests

Using Python's unittest library, we have created unit tests to verify that the models in the production database will behave correctly. Testing the production database itself is dangerous, because the tests can alter the information stored in the database if written improperly. Because of this fact, we use Django's version of the unittest library to create a small testing database that is created and deleted upon each execution of the test file. The testing database contains simpler versions of a few objects that exist in our production database.These tests utilize various queries we might make upon the database and compare the results to the predicted behavior. The models must be able to be filtered into smaller query sets, link to other types of models appropriately, return the correct string when the "str()" function is called, and have many other useful functions.

test_str_<*model name*>():

The first three unit tests simply verify that the string function for each model works. These tests use the "get()" function with the name of the desired object as a parameter in order to find that object in the table for that class. They then test the equality of the name given to "get()" and the name returned from "str()" on that object.

test_foreign_key():

This test gets a region by name. It then gets a recipe that belongs to that region by using the region object as a parameter for "get()". Only one recipe will belong to each region at this point. The test then checks that the name of the recipe is the name of the only recipe that should belong to that region.

test_filter_contains():

This test gets a recipe object by name, filters each object that has that same name into a queryset, and verifies that the queryset exists. It is helpful to know that a queryset does not exist if it is empty.

test_field_contains():

This test uses filter() to get a queryset of all recipe objects whose name contains the word “chicken”. The “icontains” in the parameter for the call to “get()” specifies that “chicken” is case-insensitive. The test then verifies that a recipe whose name is “Southern Fried Chicken” is in the queryset. Testing this behavior verifies that we can create querysets of similarly-named objects, regardless of where in the name the common word is.

test_exclude():

This test creates a queryset which should contain all recipe objects that do not contain the word “chicken” (case-insensitive) using “exclude()”. It then verifies that no recipe with the name “Southern Fried Chicken” is in the queryset.

test_incorrect_lookup():

This test attempts to get a chef using the name of the corresponding chef’s recipe. The test then verifies that no such chef exists. Testing this behavior establishes the fact that objects cannot be confused with objects derived from related models.

test_multi_field_filter():

This test creates a queryset using “filter()” and two different field constraints. It then verifies that the only recipe that should be within both of the constraints is, in fact, in that queryset.

test_<*model name*>_<*model name*>_relationship():

These three tests examine the various one-to-many relationships between the models, drawn out in the UML diagram above. They create a new object to artificially establish “many” that should belong to “one”. This new object is assigned a foreign key that matches to an already existing object. The tests then create a queryset, filtering the set of all objects that are the “many” in the relationship by the name of the “one” object. The tests then verify that the correct objects are in this queryset. After the verification step, the tests then delete the new, artificial object so that the database is returned to its original state.

API Tests

test_<*model name*>_api_1

The three tests that follow this format access our API by using the Python library “requests”. We used the requests.get() function in order to receive a response from our api. The URI sent through this call was of the format: “/http://regionalfoods.pythonanywhere.com/api/<model name>/{pk}/”. These unit tests were created in order to check that if a valid URL with a valid primary key was sent to our site, it would return the correct response. In all cases we check to see that the response returned a 200 status code and that only one item was returned. Furthermore we made sure that

there was a “fields” key whose value was a dictionary, and inside that value was a “name” attribute.

test_<*model name*>_api_2

For the second group of three tests for the API we tested the functionality of an API call that didn’t include a primary key. The correct result is a list of dictionaries representing all of the objects for the model specified in the URL. We used the same library and method as in “test_<model name>_api_1” in order to access our api through Python. In this case we sent the URL in the format of: “/http://regionalfoods.pythonanywhere.com/api/<model name>”. The test receives the response from calling the API and checks the status code to ensure it was a 200. We then check the length of the payload to make sure that the list that was returned was of length 10. Lastly, we choose an element from the list, and check so that it is in the right format with the same logic used in “test_<model name>_api_1”.

test_<*model name*>_api_3

In the last three tests written for the API we sent requests which had an invalid primary key. The correct response has a 404 status code. We used the “responses” Python library by sending the “http://regionalfoods.pythonanywhere.com/api/<model>/300” URL as a parameter for the get() method. In this test we simply check the status code to confirm that it is in fact a 404.

Search Tests

test_normalize_search_(1-3)

Three tests were written in order to test the functionality of the normalize_query() function which is used to parse a given query but stripping whitespace, grouping words that are surrounded by quotation marks, and putting single words into a list. The first two of the three tests pass queries that may be seen. For example double quotes, spaces, and other characters are used to fully test the range of things the function can do. The last test sends a query with only spaces to make sure that the result is an empty list.

test_get_query_search_(1-3)

In this series of tests it was hard to check the output of get_query() since it returned a complex “Q” object, therefore we decided that it was a better idea to use the “Q” object in a filter and check the results. With the three tests we used specific queries so that a different model was used in each one. We then checked the list returned by filter to make sure that there was one item and that the item in the list had the name we expected to be there.

test_search_search_(1-3)

These three tests make use of the search() function which is the call that the search bar on our website uses. For these three tests we used the same queries as in the test_get_query_search_(1-3) unit tests, but unlike get_query(), search() returns a much larger dictionary which has all of the results instead of half of the results for one model.

Conclusion

Despite any hiccups in the course of building RFIDB, we are satisfied with our final implementation. However, we expect the design to change eventually. While the design of the website achieves a desired “modularity” (the comprehensive use of Bootstrap columns), the overall composition does not tie in to the theme of food as well as we would like; although, such a desire might be too much to ask for without becoming visually unappealing.