Regional Foods IDB

Witty CS Pun

James Bradford

Calvin Aisenbrey

Sammi Cooper

Binwei Lin

Jishen Li

Anthony Lopez

Introduction

Regional Foods IDB is a (soon-to-be) 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 only the first iteration of the project, the website is not exhaustive in terms of available recipes, regions, and chefs. Later iterations should rectify this problem, but as the website stands, it functions fully in the intended manner. Because the web pages are static, the website does not facilitate the introduction of new data. In the future we hope to design dynamic pages so that RFIDB is not restricted and may expand in content.

The project required that we design a RESTful API, a set of django models corresponding to that API, and three static HTML pages for each of our models. Since the models and API were not required to be connected to the website in this iteration, we decided to start by building the

website first as this seemed to be the most daunting of the three overarching tasks. Such an idea proved to be fruitful as the web design portion of the project took the majority of our time.

Our group experienced a slew of organizational problems due to such issues as lack of experience with the tools required for the project, obligations to jobs and other classes, transportation, and general confusion with regard to the expectations of our submission. Many of these issues were resolved with the use of Facebook chat, Google Docs, and active members who had much more free time than others. In addition, an estimated one-third of total project completion time was spent learning the nuances of HTML, CSS, Twitter Bootstrap, Django, and sleek/intuitive web design.

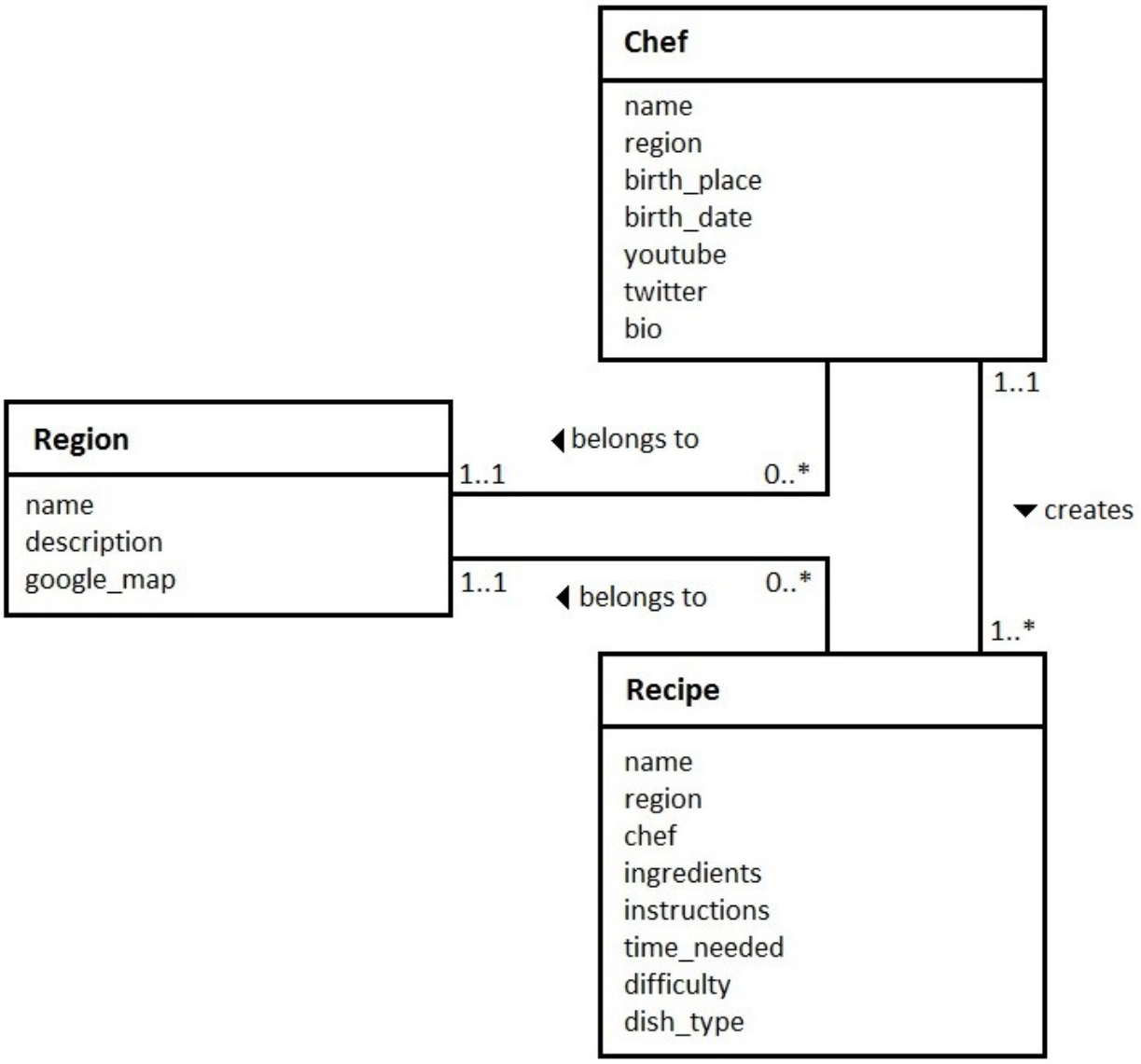
Django Models

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 will later be used to create objects that will 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. 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 other objects from other models in our database. 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. The fields that

contain foreign keys will be used to link to other pages on our websites. 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.

The Region model contains a character field for the region's name, 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. The Chef model contains a character field for the chef's name, 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 information for an embedded Youtube video, a text field that contains information for an embedded Twitter feed, and a text field that contains a brief biography for the chef. The Recipe model contains a character field for the recipe's name, 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 recipe's difficulty (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). Each model class 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.

UML Diagram for Django Models



Unit Tests

Using Django's extension of Python's unittest.TestCase, we have created unit tests to verify that the models in the database will behave correctly when actually implemented. At this phase of the project, the unit tests cannot actually be run because we have not yet created the database.

However, we can predict the expected behaviors of the models. 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.

The first three unit tests (whose function names begin with “test_str”) 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.

The next test, “test_foreign_key()” 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. The test, “test_filter_contains()” gets a recipe object by name, filters each object that has that same name into a queryset, and verifies that the queryset exists (a queryset does not exist if it is empty). The test, “test_field_contains()” gets 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. The test, “text_exclude()” 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. The test, “test_incorrect_lookup()” attempts to get a chef using the name of the chef’s recipe. The test then verifies that no such chef exists. The test “test_multi_field_filter()” 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 that queryset.

The next three tests test 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.

Static Pages

We wrote static pages in order to build a foundation for the finished website. These pages will later be replaced with django templates. Therefore, instead of having a page specifically built for each chef, recipe, and region, we will instead have one template for each chef, recipe, and region. The greatest challenge for building the static pages was learning the tools that we would need in order to built the site. Most of us were, at best, vaguely familiar with HTML, barely knew Twitter Bootstrap, and had no knowledge of Django. For several hours we researched the tools we needed and finally decided to build drafts for the static pages. Learning HTML was as simple as we had thought it would be, so it did not take much time. From there we were able to understand the Twitter Bootstrap instructions. On the Bootstrap website we learned how to use its grid system, which would allow us to format our pages exactly how we envisioned them. After employing a trial-and-error strategy on some drafts of our pages we were able to understand and utilize Bootstrap and HTML in order to build a professional, and visually pleasing website. The last tool, and the most challenging, was Django. All of us had only a month of experience in Python, and none of had ever used PythonAnywhere, let alone Django. After many hours of research and many tutorials we were able to set up Django to render the draft pages we had built.

Throughout the process of building the static webpages 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 main 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. Upon attempting this, we soon discovered that we needed to crop the photos so that they would stay at a good resolution, and their size would stay consistent. After cropping the photos, we started to make sure that the website looked good on mobile devices as well. This showed us that the floating bar at the bottom of our page covered up some of the information including a button used to navigate elsewhere on the website. We fixed this problem by adding a row at the bottom of the page with a 100 pixel margin. This would ensure that at any resolution or screen size, the page would display correctly.

After refining the index.html page with small adjustments, we moved on to building the three pages which would navigate to specific items for the group represented by each page. 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. At this point we then had to start finding data for each chef, recipe, and region. Finding consistent pages for this information was challenging, but after a little research we had the data we needed. After inserting plain text describing the chef, we decided to add an embedded youtube video at the bottom of the left column. Since we had such a small amount of experience with HTML we had to research how to add an embedded link to our page. In addition to adding the video, we decided to also include a twitter stream for each chef. This part was more challenging than the previous because the process was a lot more involving. Not only did we have to include HTML, but also JavaScript, which we also had very limited experience with, in order for the stream to display correctly. After completing those two goals we were finished with the chef template, and that was left to do in order to complete the chef pages was to copy and paste the HTML and insert specific data for each chef.

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.

After a overcoming the challenge of learning unfamiliar tools, designing a website for the first time, and deciding which material would be on the website, we built many static HTML pages which accurately show the user the information we wanted to present. By utilizing Twitter Bootstrap our website is easy to look at, but still presents our ideas concisely.

RESTful API

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. Instead of providing PUT, POST, or DELETE, we manipulate the resources ourselves inside the database that will exist later. Our API currently does not function as it is intended to, as we have not actually implemented the database from which it will get resources. The GET functions we have defined in our API will be implicitly called when a URL is specified and will provide the information specified in the body of each GET response(called a payload) as JSON.

The API provides a way use GET to list all objects of each page type (i.e. region, chef, recipe as "/<page-type>") as a 200 response with JSON, and that object has "name" as the keys and the name of each region, chef, or recipe as values. The API also provides a way to get any object by its name (as "/<page-type>/<object-name>"). When GET is called on a valid object, a 200 response is returned, meaning that JSON is returned as the response body. This dictionary 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. In the body of each "by name" response, there are actually a number of

dictionaries, one for each object of that page type. The dictionary that has a name that matches the name specified in the GET call is returned by the call. We also specify a 204 response, which will be called when an attempt is made to GET all of a page type or a specific page type that does not exist in our database or in our API.

Conclusion

Despite any hiccups in the course of building RFIDB, we are satisfied with our current implementation. However, we expect the design to change dramatically in the next two iterations of the project. 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.