Zach Abel

Carl Bai

Taylor Barnett

Joey Caudill

Andy Chow

Fried Chicken Report

Introduction:

Most young adults or college students find it difficult when adjusting to a life on their own. While some have been cooking in their homes for years, many have yet to learn how to cook more than a handful of recipes that they can actually make on their own. Once they move outside of their dormitories and cannot rely on the cafeteria food every day, they have to figure out how to cook nutritious meals for themselves. Many avoid this by just going out and buying food or precooked meals every day, but this can become expensive and/or unhealthy. Some have learned to cook a little, but this does not stretch past the stereotypical ramen. They need some way to find a few easy-to-learn recipes that they can cook on a regular basis. However, while they can google recipes and have access to a wealth of helpful resources on the Internet, they may get overwhelmed by the total number of possibilities. Additionally, most of these websites are also geared towards an older or more experienced audience. They have recipes that most young adults are simply not interested in, let alone capable of making. Some may include

ingredients that many young adults have never even heard of. There needs to be an easier and more effective way for them to learn new recipes that they can cook on their own.

Our website provides a place where college students and other young adults can go to find new recipes based off of ingredients and cuisines that they are interested in. A place where they have the opportunity to just look through different recipes knowing they are capable of making them and will be able to get ahold of the ingredients that they will need. If they do not know what an ingredient/recipe/cuisine is they can easily see what it looks like, have a useful description of it, and even be able to watch a YouTube video on it. The cuisines included provide a variety of different flavors and preferences that allow for any young adult to broaden their cooking experience and knowledge. The recipes included allow for them to create a simple yet interesting dish for each of the different cuisines. The ingredients included correspond to those in the different recipes, providing them with an opportunity to learn more about different ingredients commonly used in different types of cuisines and different dishes. However, they are ingredients that normal college students and young adults have easy access to. These three pillars (ingredients, recipes, and cuisines) provide different places for people to start when looking for what to cook. Our website provides an easy solution to many problems that young adults are commonly faced with when searching for what to cook. Suppose you have a certain ingredient lying around the house or one that you have always wanted to use, however, you cannot figure out what dish it could be used in or even what type of food it is commonly found in. Suppose you want to quickly find an easy, general recipe for a certain type of cuisine you are either interesting in learning more about or one that you are just hungry for. One thing you might want to do is to make sure it is not too complicated or exotic that you would not be able to cook the

dish itself. Suppose you just want to try a new recipe that you can easily make and add to your collection of known recipes, or perhaps you were not even looking for a specific recipe and just wanted a place to go to in order to learn about new cuisines or ingredients that you are interested in. Simply looking on this website can provide easy and quick solutions to all of these problems. This website gives young adults an opportunity to learn more about different types of ingredients, cuisines, and/or recipes that they may not have ever been exposed or known about before. It allows for them to properly adjust to a life where they can cook interesting, nutritious meals for themselves and never again have to rely on parents, takeout, and ramen.

RESTful API:

All of the data in our website's pillars (ingredients, recipes, and cuisines) are made publicly available as resources a client can access with our web REST API. The API services HTTP GET requests sent by the client and returns a JSON object containing the status of the request and the requested information back to the caller when the request is successful. For each pillar, the client is able to make two types of requests. One to get all of the data in that pillar, and another to get a single instance by specifying with an integer id. The API provides read-only access and does not respond to any other requests like POST, so the caller is not able to make any changes to the state of our resources. The root URL to access our API is http://104.239.168.220/api/v1.0. We decided to include the version of the API in the URL to make it easier to make changes under a new API version without adversely affecting existing applications that use an older version. Each pillar is accessed by specifying the name of the pillar

in the URL, and a request for a specific instance will have an additional id parameter at the end of the URL.

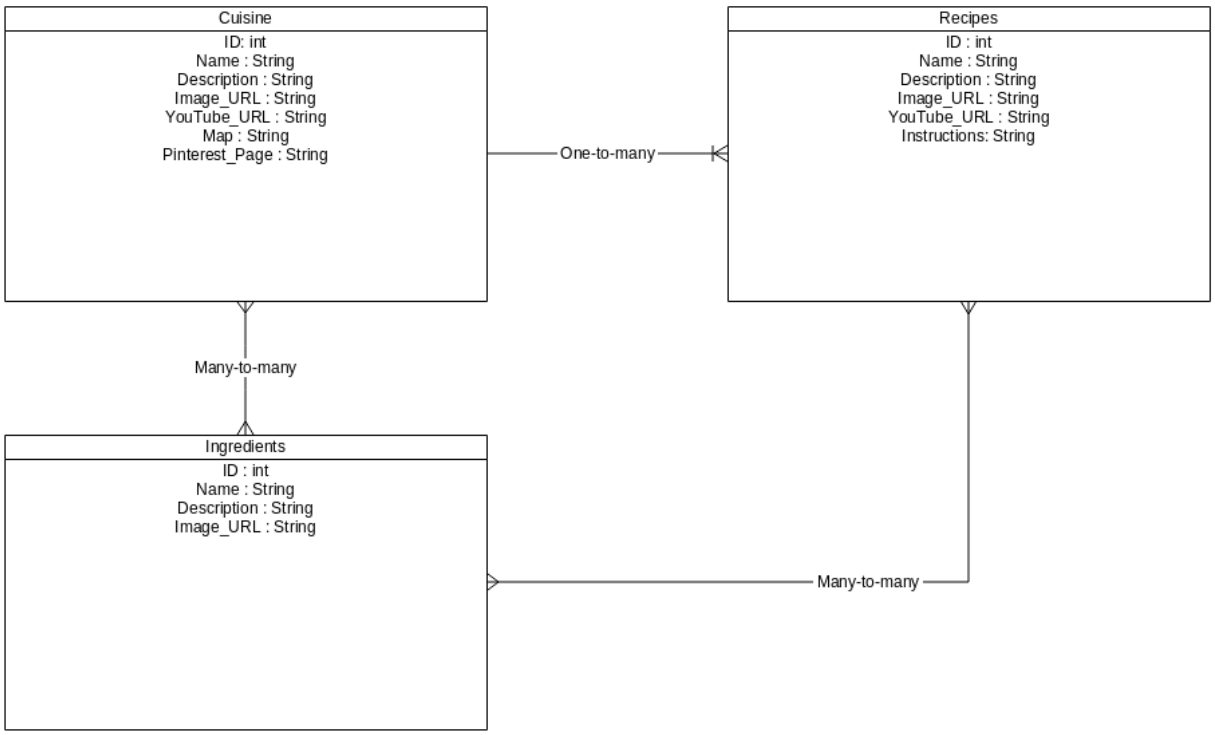
Within the responses, each response contains a status key with a value of either 'success' or 'error.' This format was adopted with a standard called JSend in mind. You can learn more about JSend here: http://labs.omniti.com/labs/jsend. The purpose of the status is to allow the user to quickly see if they were able to query the API successfully and correctly retrieve the data. When a user sends a HTTP GET request to our API correctly, we return a JSON dictionary object that contains the status with value "success" along with the data they requested, like information on a specific cuisine. Successful responses contain a data key, which is what the client will use to access the information. For example, http://104.239.168.220/api/v1.0/cuisines will return a JSON dictionary object with a JSON array object containing all instances of cuisines in our food database, with each cuisine being represented as a dictionary, as the value for the data key, and http://104.239.168.220/api/v1.0/cuisines/{id} will similarly return a JSON dictionary with a single JSON dictionary representing the specific cuisine that has the matching integer id as the value for the data key.

However, the user may query the API out of bounds by searching for a particular ID that does not exist, or the user may send a HTTP GET request to a pillar that does not exist. If the user sends a HTTP GET request that was unsuccessful, we return a JSON object that contains the status "error", an error code, and an error message. We define the error handling in `app.py` by defining an `@app.errorhandler` that handles 404 errors. The response to the incorrect GET request contains the error code of "404" and an error message of "Not found". This is used to inform the user of the error and give some information about the error.

We chose for our API to return JSON formatted responses from its GET requests for a few reasons. When using JavaScript on the front-end and calling a RESTful API, JSON is quite easy to parse because of its simplicity and helps create happiness between front-end and back-end engineers. The key/value pairs of JSON make it easy to pull information out of the response data from the GET request and embed it into the front-end code of a web application. JavaScript can natively read that data and deserialize it into objects. JSON is also a lighter structure and less bloated than XML, so it is easier to parse and can be processed faster. Occasionally if the data structures are more rigid XML is preferred, but we did not see that ours' were very rigid. Also, many modern web APIs are now all returning JSON. JSON has becoming quite popular and widely adopted as the preferred format of choice by API developers and users.

We decided to return responses with a helpful function that is a part of the Flask framework called `jsonify`. This function returns a JSON representation of the given arguments with an application/json mimetype. This is helpful for the end user of the API, so they know how the response is formatted and its type. Otherwise, it could be misinterpreted. It is also vital if it is a true REST API. Although we structured our data before the function call, `jsonify` can also structure the data for you if you pass in arguments to it. You can find out more about `jsonify` at: http://flask.pocoo.org/docs/0.10/api/#flask.json.jsonify.

Flask models:



Our website has three pillars: ingredients, cuisines, and recipes. In our model, cuisine and recipes have a one-to-many relationship, meaning that one cuisine can have many recipes, and each recipe belongs to one and only one cuisine. Recipes and ingredients have a many-to-many relationship. This means that one recipe has many ingredients and that one ingredient can be used in multiple recipes. Finally, ingredients and cuisines have a many-to-many relationship. Though some ingredients are only used in certain cuisines, there are ingredients that are more common and are shared among multiple cuisines. Therefore, each cuisine has multiple ingredients and each ingredient can be used in many different cuisines. This relationship can also show that certain cuisines have certain distinct associated ingredients.

Each of these pillars shares these following attributes: ID (primary key), name, description, and image URL. We believe that these are necessary for each of the pillars since it is important to have an ID in order to have the option of retrieving a specific item with an API call, a name that acts as a user-friendly (human readable and understandable) term for the specific item, a short description to list some useful details of the particular item, and an image to better show a visual representation of the item.

Cuisine and recipes have an additional link to a YouTube video to help the user find out more information about the cuisine and recipe. This can be useful since a video can often clarify certain things better than words. For example, a video demonstration of how to do a certain technique in a recipe would be very useful. Recipes, on the other hand, have a unique attribute called Instructions. This is an array of Strings that lists the steps required to make a specific dish. Cuisine also has the unique attributes Map and Pinterest. Map is a link to a Google Map that shows the particular location of the cuisine. This helps in understanding the geographic region a certain cuisine is from. The Pinterest attribute is a URL link to a Pinterest Board for a certain cuisine. Linking the Pinterest Board allows the user to find a community that also enjoys that certain cuisine and be able to see pictures posted by Pinterest users that are related to that cuisine, like recipes that others have tried to cook.

We currently do not have a database backend to represent this model. For the sake of simplicity, we decided to represent our model in JSON formation in `app.py`. Since we decided to return JSON objects when a user sends a HTTP GET request on our API, this allows us to use the model to directly serve API requests. This required us to write all of the collected data in a JSON format and hard coded it in `app.py` by hand. The downside of this is that we have to

define certain relationship within the JSON object. For example, we have to define all the ingredients that a cuisine uses and all the recipes it is associated with. With a database, we would be able to use database relationship to retrieve this information dynamically instead of defining it by hand. In the future we want to transition this model into an actual database that will return JSON object when requested.

Application structure:

As seen below, the application structures follows a pretty standard format as suggested by the Flask documentation. For larger applications, it is suggested for there to be another directory layer, but since our API and application is quite small we chose to keep it simple for now. Everything for the application goes in this cs373-idb folder. All of the Python code and server logic gets written into the `app.py` file. There's a static subdirectory which contains another subdirectory for the front-end's CSS, and a separate templates subdirectory for all of the HTML templates for the front-end. Also in the static subdirectory is some of our images. Currently, we do not store a lot of our images, but in the future we hope to store those within our database and server from there. These are all nested within /ubuntu/cs373-idb inside the home directory of the Rackspace Linux distro.

```
cs373-idb/
      /app.py
      /static/
            /css/
                  /bootstrap.min.css
            /image.jpeg
            ...
      /templates/
            /base.html
```

```
/index.html
...
```

The purpose of these subdirectories is to be able to better organize certain files together and be able to quickly find them when you need it. Also, it helps make it easier to render templates for example when you know all HTML templates are located inside of the template subdirectory. In the future, it may be better to create more subdirectories within the subdirectory to better organize the files.

The application is served over nginx which redirects all incoming HTTP requests on port 80 to the localhost on port 8000, where Flask is listening, and vice versa. The Flask `app.py` renders HTML templates from the templates folder with calls to get static images and content from the static folder.

Front end:

To translate from model to view, our website makes use of Flask's built-in Jinja2 templating capabilities. Right now our templates pull their data from the model (which is possible and easy since the `render_template` functions are called in `app.py`, where the model is concurrently defined with the controller), in the form of python dictionaries defined in `app.py`. Templating allows us to make use of inheritance when designing web pages while additionally reducing the effects of networking latency as opposed to pulling content from the API with JavaScript XMLHttpRequests. Right now our inheritance hierarchy is fairly shallow; we only inherit the headers and footers from a base.html file.

Other front-end technologies include Twitter Bootstrap to layout our content, which allows us to easily create mobile friendly pages that can nevertheless take advantage of screen

space via Bootstrap's grid system and relative positioning capabilities. All of the site's content is contained in Bootstrap's jumbotron class, which allows us to have multiple background colors which helps to emphasize the content and shift attention from blank space. To select colors we used the simplex Bootstrap theme. Additionally we provided a navigation bar to keep the site's three pillars easily accessible from any page on the site. The navigation bar makes use of Bootstrap's JavaScript to provide a pull down animation for narrow screens.

HTML `<a></a>` tags were used to make as many elements of the site linkable as possible - within reason - everything from titles to pictures to ingredient lists references other pages and columns within the design, both to encourage use of the website and enable ease of navigation and traversal. We used a dirty hack in the form of empty `<p></p>` tags on the teams page to put space between each team members' profile; the only slightly less dirty `</br>` tag failed to work when it unevenly applied vertical spacing to the second and third profiles.

Unit tests:

For our unit tests we made use of requests, which is an HTTP library written in python. It allows for you to simply make HTTP requests on a given URL in order to get back the the web page in the form of a Response object. It does all the hard work for you. So after importing the requests module, you can use the function requests.get to get back a Response object on a given URL, using the form `r = requests.get('http://some.url')`. For our unit test the URL is just the URL for the given route we are testing, either on our web API or our website. Then we can check all the data on the given Response object, which corresponds to the given URL.

In order to see if our Response object we got back was correct, for each test we made sure to first test that the type of the response was correct, using `c = r.headers['content-type']` to get back that type. Then you can continue to use the Response object by checking the content of the object itself. For API route tests we used `r.json()`, which decodes the Response object into a JSON file, as long as it is a JSON file. We then checked the different attributes within the JSON file to make sure the values in the were correct. For the web routes we had to check the contents of the HTML page. So, we used `r.text` to get back a long string representing the HTML of the page. Once we had the HTML we just checked if a certain substring that was unique to each page was located within this long string, using 'substring' in `r.text`.

We made use of the python unittest framework, using `assertEqual` in order to see if the content, status, type, etc. were all correct for the given request, as well as `assertTrue` to see if the content was correct for the HTML pages. In our unit tests we test normal cases for each API route and each website route, as well as certain edge cases and error cases. Commonly tested error cases include badly formed URLs, bad input for specifying on a pillar, and out of bounds on number of elements of a pillar.

Conclusion:

Although we are in a great place to start, we still have things to add in future parts of the project. Since we chose a more simple route for "models," adding the database layer and more formal database models along with that will take a decent amount of time. While at the same time, we did not do a lot of work to get the models we have now, so if we decide to make any

changes before we implement a database layer it will not have a ripple effect in other areas of our code.

Currently, we are using almost no JavaScript in our site. This is perfectly fine since we did not need to call our own API yet. In the future, we will research and decide on a front-end framework to help facilitate that.

In a lot of our design decisions, we chose simplicity and functionality so we could make progress in the project. We knew that things might change as we go on so this simplicity will help us not waste time on code we might abandon. In the next step of the project, we will increase the complexity but will have a better idea of how we want things.