RecipeFinder *name still in the works*

Our team members are Cheng Xi Tsou, Ningyu Chen, Azad Ebrahimian, and Sukhman Singh. For our final project, we decided to create a web application that allows users to research recipes based on ingredients that they already own. Rather than having to mindlessly look through your fridge and cabinets to guess what you can cook, our web app will allow you to find the perfect recipe based on those ingredients. In the web app, a user will be able to have their own virtual “fridge.” This “fridge” represents the list of pantry and fridge ingredients that a user already has and is willing to cook with. At any time, a user is able to update their virtual “fridge” with any new ingredients they acquire or ingredients that they have run out of. In order to discover possible recipes for the user, the web app will use an API and send its list of ingredients. The API will then return a list of possible recipes that require the ingredients existing in the user’s “fridge.”

The goal of our project is to simplify the awkward struggle some people have when it comes to making food with the ingredients they already have. When you have so many random ingredients sitting around your home, it sometimes becomes a hassle picking and choosing which to use, and what are the possibilities to cook with them. Now, with our project, this hassle is eliminated; and users can discover all the possible recipes that were hidden from them when blankly looking through their pantry.

The API we planned to use is called Spoonacular. As our project is about getting recipes given a list of ingredients, we needed a good API that can allow us to easily get a list of recipes from a list of ingredients without going through too many API endpoints and having to create too many relations in postgres. Another concern of ours was the pricing and API call quotas. Ideally, the API should be free but usually the free API services are very limited in which endpoints we could use or there is a low ceiling for allowed API calls. Lastly, we need the API’s database to contain enough recipes and information so that our web app is attractive enough to users. As there are so many ingredients out there, there is an infinite number of combinations of ingredients someone could have, and getting a list of recipes from that is a hard task. Spoonacular provides a very generous plan that includes 150 points per day for free, with one call costing 1 point and every result returned costing 0.01 points. The endpoints that Spoonacular provides is sufficient too. You can get information about a specific ingredient or recipe given its id, which will be useful as we can just store the id of each recipe and make a API call every time a user needs more information about a recipe. This would reduce the data we would need to store in our database as we can just parse information from the API call instead of making a query. However, there is a tradeoff between having less complex relations and needing to make more API calls, which can hit the daily limit pretty fast. Spoonacular also provides an easy way to get a list of recipes that you can make given a list of ingredients. There is an endpoint that returns a list of recipes with information about used ingredients and missing ingredients, and the API does try to minimize the number of missing ingredients.

A big question that came out when trying to choose the right API for our project was the list of ingredients itself. Did we want the user to type in their own ingredients or have a preset list? Letting the user type in their own ingredients would give more freedom to the user, but it would require more work to validate the inputs; as we are using an API endpoint to find the list of recipes, it would also require a preset list of ingredients to choose from. A problem about using a preset list would be that there aren't enough ingredients on that list, which would ruin the user experience. Luckily, the Spoonacular API has over 2,600 ingredients in their database, and there is a downloadable csv file of the top 1,000 ingredients provided with the ingredient name and associated id.

The real-time feature we will add to our project will be a status to each user. Users could change this status to "Cooking", which would mean that they were currently cooking. This would notify all of the user's friends (on the web app) that they were cooking at that moment, introducing a real-time aspect to our program. It also makes it more interactive with other members. If a member is notified of a friend cooking, we hope it would either encourage them to find a recipe and cook or maybe even contact that friend to learn more about what they are cooking. The other possible states of users will be "inactive/offline", or "online". The former means a user is not currently logged into the website, the latter means they are logged in but are not currently cooking. To bolster the interactivity of the website, we will consider adding a messaging feature that allows you to communicate with other friends in some capacity. Depending on how intense the other portions of the assignment are, this could mean simple pings with pre-made messages or a full blown text chat feature.

The goal of this web application is to build a cooking community among young adults. It assists the users in managing their ingredients in the refrigerator and connects with other users. The following is a description of our stage-one SQL database design in terms of "entity". Each **user** has a unique username on registration, a name(first and last), and a password of some restrictions. The password will be stored after salting and hashing, so we also need to store the salt. A user can have multiple favorite **recipes** and recipes that are available according to their ingredients, so he/she will also have multiple **owned-ingredients**. Since we are building a community, a user can make **posts** and friends with other users. A user can also blacklist other users. A user can make **comments** on posts of friends and public posts. A user can **like** or display each post. A user can have a profile picture. Each **friend-request** has a unique combination of receiver user id and sender user id. Once a friend-request is accepted or rejected, it should be processed and deleted from the database. So we are just storing the time when a request is sent. Each **friend** has a unique combination of two user ids. The purpose of this table is solely to track whether two users are friends or not. Each **black-list** has a unique combination of primary user id and secondary user id, in which the primary user is blacklisting the secondary user. Each **recipe** is associated with a **use**r by having a foreign key to the **user** id. It has a string for the API query(that queries the recipe), and a flag to indicate whether this recipe is marked as favorite by its associated user. Each **ingredient** has a name. The purpose of storing the ingredients is to prevent the user from inputting unrecognized ingredients. Maybe, it can have multiple possible measuring units. So we might need a unit table. Each **owned-ingredients** is associated with a **user** and an **ingredient**. It must have a numeric quantity(can't be zero). Each **post** is associated with a **user** by having a foreign key to the **user** id has a foreign key to a user id, a string for the API query(that query the recipe), a flag to indicate whether it is visible to the public, a picture of the food, and the time it is posted. Maybe we should allow each post to have up to 8 photos. In that case, we might need a separate photo table. Each **comment** is associated with a **user** and a **post**. It has a comment of length 100 words, and the time it is created.Each **like** is associated with a unique combination of **user** and **post** by having references to their primary key and uniquely indexed. It has a flag to indicate whether it is a like or dislike. To support (public and friend exclusive) live feeds, we might need to have a table to track logged-in users and their socket id or channel(require more research about web-socket). Each **online_user** is associated with an online user, and it has some information about the connected socket. After implementing our stage one functionality, we might add real-time messaging functionality since our goal is to build a friendly cooking platform for young adults. Therefore, more tables will be introduced, but it has very little probability of changing our current design of the database.

A "neat" feature we will implement is allowing users to peek at other users' ingredient lists and request items. This allows for more interactivity among users and could help if a user almost has all the ingredients for a recipe but is missing one or two that another friend has. To have some privacy, users can only see the ingredients list of their friends. When a user sends out the request for an ingredient, the receiver of the request will have a choice to accept or decline it. If they choose to decline it, nothing happens and both users' ingredients lists remain unchanged. If the receiver chooses to accept it, the ingredient is removed from their list and added to the requester's list.

For this experiment, we wanted to try out the API to see what kind of information we can get from different endpoints, and how we can go from user input, parsing the API call output, to displaying relevant information for the user. As mentioned in experiment 2, we decided to create a preset list of ingredients available from the API for the users to select. Our PostgreSQL database will store the preset list of ingredients, and the user input will come from there. We wanted to see how we could go from that list to a list of recipes, if we needed to store any associated ids that the API requires, and how we could get more information about a specific recipe.

Upon investigating the Spoonacular API, we found an endpoint that takes in a comma separated list of ingredients and returned a list of recipes. This was great news as we didn't have to store an associated id with an ingredient, though we could if we wanted to show the user nutritional information about an ingredient as a potential feature of our webapp. The API call returned an array of recipes with parameters such as id, image, missedIngredients, usedIngredients, and unusedIngredients. As there was an endpoint for getting information about a specific recipe given the id, all we needed to store was the recipe id.

Another endpoint we investigated was getting information about a specific recipe or ingredient. As we needed to display information about the recipes or an ingredient if we chose to implement that, we needed to make sure we had the relevant information. The endpoint for the ingredient gave us the name, id, image, possibleUnits, and even cost, which we could possibly implement as an additional feature. As for the recipe, it gave us all the relevant information that we needed. One result we looked for in this endpoint was the measuring units of each ingredient. Since we wanted to store the quantity of each ingredient in the user fridge, this was an important piece that we needed to track.

Lastly, as expanded upon in experiment 2, there is a downloadable csv of the top 1000 ingredients in the API's database. While a preset list of items is great, a user might want to input a more obscure ingredient into their fridge that is not on the preset list. There is an endpoint for searching for existing ingredients in the API's database.

After experimenting with the API, we learned about some adjustments we needed to make about our project. For the API call to get the list of ingredients, there was a potential problem. A user could have an arbitrary combination of ingredients which there is no recipe for, so that would result in the API call returning a list of recipes that minimizes the missed ingredient. A solution would be to store the missed ingredients list in our database.

Another point of adjustment we would need to add to our project would be how we tracked the quantity of ingredients in the user's fridge. As there are no universal units for each ingredient, we would need to pick a unit or have the user choose a unit to track their ingredients by. The endpoint for getting ingredient information does provide a list of possible units, so we could have the user select a unit from that list. However, from our experiment with the endpoint for recipe information, different recipes may have different units for the same ingredient. Luckily, there is an endpoint for converting units, so we can automatically track a user's ingredient quantity when a user "cooks" a recipe food item.

To utilize our API, we decided to store a list of ingredients that can be chosen by users instead of having users type in arbitrary ingredients. This eliminates the risk of users mistyping the names of ingredients as well as reducing redundant calls to the API to retrieve lists of ingredients. To do this, we want to store a list of ingredients in our PostgreSQL database in our project. We first acquired a list of the top 1000 ingredients from our API's website (Spoonacular). This list was formatted in a CSV document. Next, we needed to find a way to convert rows of this CSV file into accessable data for our database table. Our experiment was finding a way to store these 1000 rows into our table, so that they could be accessed in the project.

We began by creating a table in the PostgreSQL console, and ran a script that would copy the data from the CSV file into that table we just created. This resulted in a complete table for our postgres user, but it was not connected to our Phoenix application in any way. We discovered that we needed to insert these items to our table in our seeds.exs file. To do this, we first generated a table for Ingredients with Phoenix. Then, we wrote a script in seeds.exs that would parse through the rows of the CSV file and individually insert each row as a record in our Ingredients table (inspired by the code presented here http://wsmoak.net/2016/02/16/phoenix-ecto-seeds-csv.html). In order to parse through the CSV file, we imported the CSV dependency (https://hex.pm/packages/csv). This script was able to successfully insert every row of the CSV file into our Ingredients table in the database, and is now accessible to our project.

This was a vital aspect of our plan because without it, we would either have to manually enter each ingredient to our database or we would have to allow users to input any arbitrary ingredient they have. The former would be inefficient and the latter would cause many new issues, such as the possibility of invalid inputs and incompatible inputs with our API.

This experiment taught us how to load large quantities of data into a pre-existing table in our Phoenix application. The method we utilized allows us to update the list of ingredients we use at any time to add or remove items. Through this experiment, we also solidified our plan on how users will store ingredients to their account. Because we successfully created a list of ingredients in our database, we know we can make a relation between user and ingredient (a user “has_many” ingredients) to effectively manage our users’ list of ingredients.

For our web app, we have two target users: parents and college students. Parents have the struggle of needing to feed themselves as well as their kids. Our web app will be beneficial to parents since they can virtually keep track of their ingredients, and efficiently find out what recipes they can cook in order to feed their families. Especially, since parents are almost always busy with some other kind of work around the home, figuring out what to cook for their families is only another chore that they may not have time to spend on. The web app will assist in this issue, and basically eliminate time that would be needed to find what they should cook for their family that day.

On a similar note, college students generally have the struggle of not knowing what to cook for themselves everyday. They also tend to have a much more limited set of ingredients to use, so finding recipes to make is a lot more of a struggle. Our web app will help students like this organize and view their ingredients virtually, and directly see what kinds of recipes they can create with the limited ingredients they own. The interactions that we support on our app will be targeted towards this user base to encourage college students to cook.