Senior Capstone Project – Final Report

Project Name

Household Menu Planner and Ingredient Tracker (HoMePIT)

Team Members

- Erik Anderson
- Jerome Chestnut
- Maxwell Fugette

Introduction

Menu planning is an inescapable and oft time-consuming part of human existence. Some individuals attempt to avoid it by subsisting on fast and highly processed food, while others try their best to manage it using whatever tools are available to organize and categorize this part of their lives. With healthy eating being the preferred option, it is imperative for it to be made as simple and convenient as possible. We believe that many of the tasks associated with meal planning, such as the tracking of household ingredients and foodstuffs, planning what meals to eat on what days, and the composition of shopping lists are often too tedious and time-consuming for the average person, which will inevitably lead to unhealthy dietary choices. All of this can be simplified with the appropriate application of technology. HoMePIT is our attempt at making this simplification.

HoMePIT, as its expansion suggests, assists primarily in the tasks associated with planning a household's menu and tracking its supply, usage, and replacement of ingredients and other foodstuffs. It is designed to streamline the culinary experience of a household by making ingredient inventorying, grocery shopping, meal planning, and nutrition tracking easier to manage on an everyday basis. Using Google authentication, users can easily sign into HoMePIT and start using it. Users are able to add ingredients, recipes, information relating to ingredients and recipes, and more to HoMePIT. Simultaneously, HoMePIT assists users in tracking nutrition, creating menus, and producing shopping lists covering user-defined time periods. With this, HoMePIT is intended to not just be an application, but rather as essential to a kitchen as the food itself.

Tools and Technologies

Platform

The platforms chosen for HoMePIT to run on were the internet browsers of Mozilla Firefox and Google Chrome. These browsers required minimal effort to familiarize oneself with, as the developers of HoMePIT use them every day and have used them in past project development. This was exemplified by the fact that no issues were encountered during HoMePIT's development that could be attributed to browser-unique behavior. Accordingly, Firefox and Chrome did not hamper development, although they did not better enable development relative to the average development experience. HoMePIT was designed with Chrome and Firefox from the start of its development and thus the platforms targeted never changed during its development.

Operating System

HoMePIT has been designed for usage in browsers hosted in the Windows and Android environments. These environments were targeted because of both the developers' familiarity with them and due to their ubiquity. Much like in the case of the browsers, it was easy for the developers to get up to speed with these technologies, as the developers use them daily. Given that HoMePIT was developed as a web application and that both Firefox and Chrome are available on Windows 10 and 11 and Android, the choice of operating systems had little impact on the difficulty of development. The greatest influence that they had was via the fact that the Android operating system appears most often in mobile form factors, which required that user interfaces and experiences be developed with the intention of being used on a mobile device. In this sense, some difficulty was caused, as development primarily occurred in desktop environments. Thus, the developers found that it was easy to neglect adjusting the UI to be perfectly compatible with mobile interfaces, and this is shown in the fact that some elements of the HoMePIT interface display poorly when viewed in portrait mode on mobile devices.

Integrated Development Environment

VSCode was the Integrated Development Environment (IDE) eventually used for desktop development of HoMePIT. This IDE was familiar to all members of HoMePIT's development team, as they had used it for prior classes, and thus all team members were able to easily begin using it for development. VSCode eased development by providing the developers with a full fledged and extremely customizable workspace. Originally, the developers had thought to use Visual Studio and Replit alongside VSCode. However, they found that all three IDEs had a significant degree of overlap in their functionalities and eventually decided that VSCode had the most desirable blend of features and familiarity. In making this decision, the developers reduced the mental overhead required to remember and track the differences and nuances to 3 different IDEs, thereby making development easier.

Programming Languages

A variety of programming languages were used in HoMePIT's development, such as JavaScript, TypeScript, HTML, and CSS. These languages were chosen due to their general necessity for web development. The developers were broadly familiar with these languages and thus were able to quickly get up to speed with them, and the languages made development easier due to the aforementioned developer familiarity with them. Originally, there had been some intention to use C# in the development of HoMePIT, as the developers had some degree more recent familiarity with C# relative to the aforementioned languages. However, this idea was abandoned because it was found that there was a larger quantity of web development resources available for JavaScript and TypeScript, and it was thus determined that it would probably be easier to develop HoMePIT using JavaScript and TypeScript as opposed to C#. In hindsight, it is possible that development would have been easier when using C#, as the developers responsible for frontend development recall that C# contained many convenient solutions to problems that were encountered when developing HoMePIT with TypeScript and JavaScript.

A final language used in HoMePIT's development was that of Procedural Language/PostgreSQL (PL/pgSQL), which was chosen on account of it being the language presented for development in SupaBase. This was a necessary language to adopt to enable the usage of SupaBase, which in turn needed to be used due to SupaBase providing a free relational database service on which HoMePIT could host its data. Given its similarities to MySQL, which the developers had familiarity with due to previously taken database management courses, PL/pgSQL was fairly easy to adopt. This ease of adoption meant that it made development relatively easy, although debugging options were limited within the context of using PL/pgSQL on SupaBase.

3rd Party Tools, Libraries, & Services

A major 3rd party tool used during HoMePIT's development was Git version control, which was provided through the 3rd party service of GitHub. Git and its usage on GitHub was easy to adapt to due to previous courses having taught the developers how to use them both, and they made development easier by providing centralized version control for non-database development.

Another 3rd party service central to the functioning of HoMePIT would be that of SupaBase, which provided necessary relational database hosting services. While none of the developers had experience with the use of SupaBase, they found it fairly simple to become accustomed to due to its similarity to other database management services and well-designed UI and UX. SupaBase also eased development by moving the challenges of hosting a database to a 3rd party, although such would have occurred regardless because of the requirement to not host one's own database. Google's FireStore service had been considered prior to the selection of SupaBase, but that idea had been discarded when HoMePIT's developers were advised to adopt a relational database instead of using NoSQL. This proved to be a good idea, as the data used in HoMePIT is highly relational.

The 3rd party libraries used in HoMePIT were React, Bootstrap, Material UI, and FullCalendar. FullCalendar was used to enable the implementation of a calendar in HoMePIT's menu, and it proved to be very challenging to familiarize oneself with. Generally, the usage of FullCalendar and resulting decisions made because of its usage made development harder. React, Bootstrap, and Material UI were selected due to their ability to provide HoMePIT with a UI and

UX that was relatively modern without requiring the development of such from scratch. While these technologies did make development easier, there was a significant amount of nuance required to use them effectively, and ultimately it took a notable amount of time to become accustomed to their use. Originally, the intent of the developers had been to use the ASP.NET framework to develop HoMePIT, as this would have allowed them to leverage their comfort with C# while enabling them to perform web development. However, for the same reasons as previously discussed with respect to C#, the developers chose to not pursue the use of ASP.NET. Similarly, they believe that ASP.NET may have been easier to use in some respects, although they are uncertain if it can be specifically stated that abandoning ASP.NET was a bad idea.

Server Software & Services

HoMePIT is hosted by a variety of server softwares and services, including Node.js, Google Firebase, and SupaBase. Node.js was selected as it permits the hosting of websites built on JavaScript, and it was fairly easy to familiarize oneself with due to previous exposure to its use. It made development easier due to its ubiquity in web development, which meant that there were a variety of resources available detailing its use. Google Firebase was selected as a hosting service due to the documentation available for it and nearly-nonexistent cost. It was easy to familiarize oneself with and made the hosting of HoMePIT easy.

Design

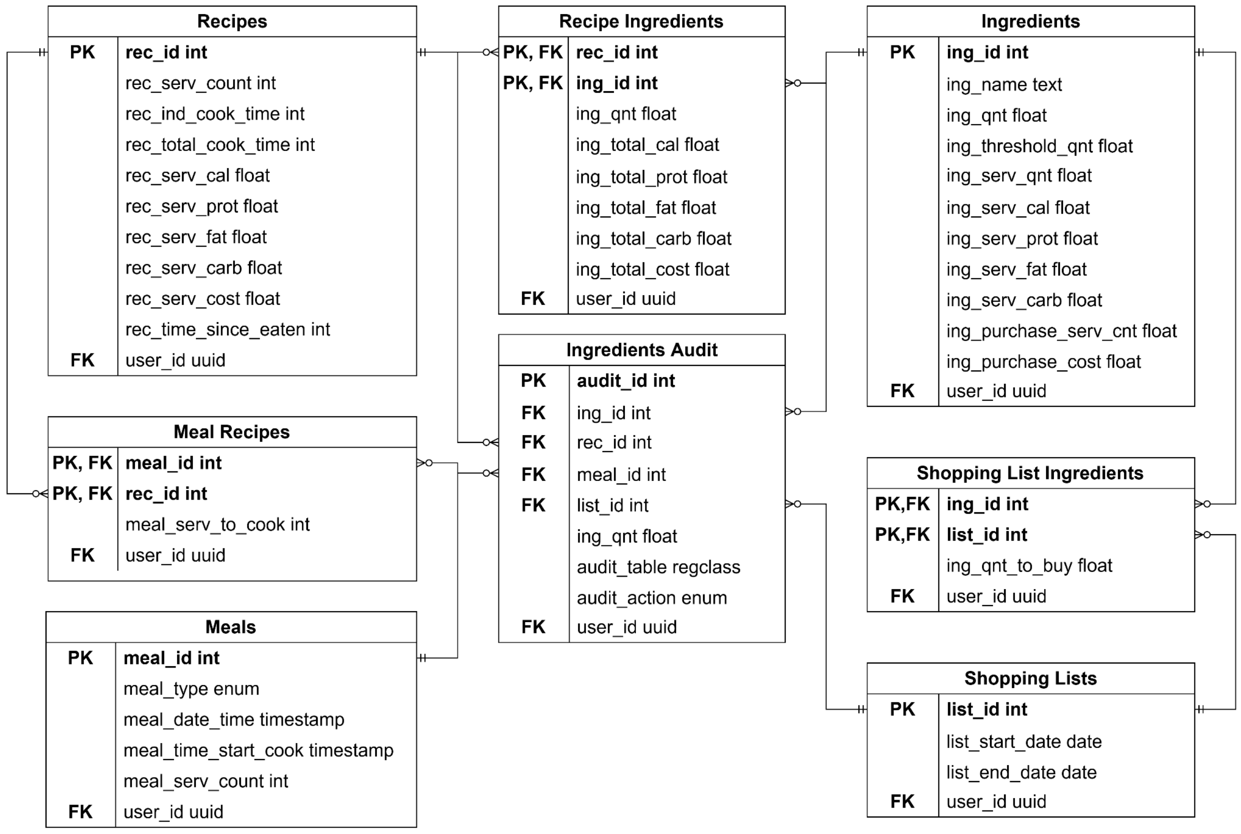

Figure 1. The final Entity Relationship Diagram of HoMePIT v1.

Database Tables

Ingredients

The Ingredients table contains information related to ingredients in a user's household. As seen in Figure 1, it has a number of columns, and in order they include the ingredients' IDs, names, quantities (measured in either mL or g, although the actual value stored is unitless and the units are implied), threshold quantities (the point below which an ingredient requires replenishment), serving size quantities (the quantity contained within 1 serving of an ingredient, as defined by an ingredient's nutritional info), calories per serving, protein per serving, fat per serving, carbs per serving, the number of servings received when the ingredient is purchased (assuming that a user always buys the same quantity of ingredient), the cost of purchasing that number of servings, and the user ID affiliated with that entry.

There are two triggers and functions associated with the Ingredients table. The first, ing_audit_on_ing, fires after a row is inserted or updated in Ingredients. It executes the function ing_audit_on_ing(), which inserts into Ingredient Audits a row detailing that a given ingredient had a certain quantity added or subtracted from it, that such occurred as the result of an insert or update on Ingredients, and time stamped with the current date and time. This does not occur on a row deletion, as when an ingredient is deleted all information about that ingredient is cascade deleted from the database. The second trigger, update_rec_ing_nutr_on_ing, fires whenever a row is updated due to a change in an ingredient's nutritional information. It executes the similarly named update_rec_ing_nutr_on_ing() function, which sets the nutritional info of any rows in Recipe Ingredients that have the same ingredient ID to the appropriate updated value using the new nutritional information and the quantity of ingredient specified in each row of Recipe Ingredients.

Recipes

The Recipes table contains information about recipes that a user chooses to store in HoMePIT. Going in order as appearing within Figure 1, this information includes the recipe ID, serving count, individual and total cook time (a bifurcation that exists as an artifact from when sub-recipes were still to be implemented), calories per servings, protein per serving, fat per serving, carbs per serving, cost per serving, time since one has eaten the recipe last, and the user ID associated with a given recipe. To adhere to the ideas of the third normal form in database table design, the Recipes table does not contain the actual ingredients in a recipe - this is instead specified in the auxiliary table Recipe Ingredients.

There are two triggers and associated functions applied to the Recipes table. The first trigger, update_rec_nutr_on_rec_update, applies the function rec_calc_serv_nutr_on_rec_update(). This trigger activates before a row update on the Recipes table that changes the serving count of a Recipe. When triggered, it recalculates the number of calories in a recipe based on the newly specified serving count while keeping the total ingredient usage the same. The second trigger would be ing_aud_on_rec_update, which triggers before a row is inserted, updated, or deleted, and executes the function ing_audit_on_rec(). This function serves to appropriately update the Ingredient Audits table should a change be made to a recipe that could impact the quantity of ingredients used in a meal. This occurs when the serving count of a recipe changes, since this will then impact the quantity of ingredients used to make a given serving count for a meal. For example, if a recipe was previously specified as providing 4 servings for a given ingredient input, but is updated to say that in reality it only serves 2, then a meal requiring 4 servings of a recipe will call for twice as many ingredients afterwards. Likewise, when a recipe is deleted, this function negates any usage of ingredients that has yet to occur (specified as ingredients used after the user's current time) and inserts the appropriate entries into the Ingredients Audit table.

Recipe Ingredients

The Recipe Ingredients table contains information about each recipe-ingredient pair that exists, with a pair existing if a given ingredient in some quantity is part of a recipe. This information includes the IDs of the recipe and ingredient in question, the quantity of this ingredient in one preparation of the recipe, the total calories conferred by the quantity of the ingredient, the total protein conferred by the quantity of the ingredient, the total fat conferred by the quantity of the ingredient, the total carb conferred by the quantity of the ingredient, and the affiliated user ID.

The Recipe Ingredients table has 3 triggers associated with it - update_rec_ing_nutr_on_rec_ing_update, update_rec_nutr_on_rec_ing_update, and ing_aud_on_rec_ing_update. When update_rec_ing_nutr_on_rec_ing_update triggers, which occurs before a row is inserted or updated in the table, it executes the function rec_ing_calc_total_nutr_and_cost_on_rec_ing_update(). As is hinted by the function's name, this serves to update the nutritional information specified by any given row of the Recipe Ingredients table. When an insert occurs, it specifies the nutritional info of the row being inserted, which is calculated from the associated ingredient's nutritional information per serving, its serving quantity, and the quantity of the ingredient required for the associated recipe. Likewise, when an update impacts the quantity of an ingredient in a recipe, the nutritional information is recalculated using the new ingredient quantity. This function ultimately allows accurate nutritional information to be determined, assuming that the nutritional information for a user's ingredients is accurate. When update_rec_nutr_on_rec_ing_update is triggered by a row insertion, update, or deletion from Recipe Ingredients, it executes rec_calc_serv_nutr_on_rec_ing_update(). This function works similarly to the previously described function, except it updates the summed nutritional information for a recipe, rather than the nutritional information related to a single ingredient. When a row is inserted or deleted in Recipe Ingredients, then the associated nutritional information is added or subtracted from the per-serving nutritional information of a recipe. Likewise, when a row is updated in Recipe Ingredients, a recipe's nutritional information is updated with the per-serving difference between the previous and current nutritional information of that ingredient. Finally, ing_aud_on_rec_ing_update triggers before row insertion, updates, and deletions, and inserts audit logs when the quantity of ingredients to be used is changed. For each case, this occurs only if a future meal exists that uses the recipe associated with a given row. On insertions and deletions, the calculated quantity of ingredient required for the preparation of a given recipe for a given meal is either subtracted or returned to the ingredient quantity. On updates, the updated difference between the previously and currently used ingredient quantities is stored in an audit log.

Meals

The Meals table contains information about each meal that a user has specified in HoMePIT, such as its ID, the type of meal (which can be either breakfast, brunch, lunch, dinner, dessert, or a snack), the date and time that the meal is scheduled, the date and time that the meal must start to be prepared, the number of servings required for the meal, and the associated user ID. As was the case with Recipes, the third normal form was more closely adhered to by having information concerning food consumed during any given meal stored in an auxiliary table joining Meals and Recipes.

There is a single trigger on the Meals table, meal_rec_on_meal. This trigger executes the function meal_rec_on_meal, which updates any Meal Recipes with the same meal ID with a new meal serving count if the meal serving count was changed. This function exists to minimize data anomalies due to duplicated data.

Meal Recipes

The Meal Recipes table specifies the meal-recipe pairings that a user inputs, which is detailed using the meal and recipe IDs that a specific pairing joins, the number of servings required, and a user ID. Some deviation from the third normal form may be argued here, as the number of servings for a given recipe at a meal is specified for each meal-recipe pairing, despite this information being contained in the Meals table. However, this is an artifact from when functionality related to leftover tracking was still intended to be added in version 1 of HoMePIT. If that functionality were implemented, then this supposedly-duplicate value would have use, as it would help differentiate when a recipe needed to be cooked versus prepared from leftovers.

There is a single trigger on the Meal Recipes table, ing_aud_on_meal_rec_update, which triggers before a row is inserted, updated, or deleted and executes ing_audit_on_meal_rec(). This function serves to insert rows into the Ingredients Audit table specifying the usage of ingredients (on an insertion or update) or reduction in usage of ingredients (on a deletion) associated with a given meal-recipe pairing. On an insertion, if the meal is not in the past, the function calculates the quantity of each ingredient used to make the specified number of servings of the specified recipe and inserts these quantities as ingredient reductions in audit logs in Ingredients Audit. The same occurs on deletion, although with ingredient additions being specified in the generated audit logs. When an update occurs, if the number of servings has changed, then the associated increase or decrease in ingredient usage is logged.

Shopping Lists

The Shopping Lists table details the shopping lists that a user currently has available. The information associated with a shopping list is its ID, the first day that its coverage of a user's meals starts (which also represents the day that a user buys the contents of a shopping list), the last day of meals that a shopping list covers, and the user ID of the associated user. Much as was the case with the Recipe Ingredients and Meal Recipes tables, the actual ingredients specified by a shopping list and the associated quantity to purchase are not specified in the Shopping Lists table to better adhere to third normal form principles.

There are three triggers on the Shopping Lists table - shop_list_on_shop_list, shop_list_ing_on_shop_list, and ing_audit_on_shop_list. When shop_list_on_shop_list triggers before a row insertion or update, it executes the function shop_list_on_shop_list(), which checks whether a shopping list insertion is valid. It disallows insertions where the start date comes after the end date, and any insertion that would produce a shopping list overlapping with any other shopping list. It also disallows the insertion of any shopping list with a start or end date starting before the current date, as shopping lists and their associated shopping list ingredients are calculated with the assumption that shopping lists have been purchased on or before their start date. With respect to updates, shop_list_on_shop_list() disallows any updates to shopping lists, as it was determined that the difficulties that would arise from having to properly track changing shopping list dates were too severe and numerous to handle in the initial instantiation of HoMePIT.

The trigger shop_list_ing_on_shop_list fires the function shop_list_ing_on_shop_list() after a row is inserted into Shopping Lists. This function serves to populate the Shopping List Ingredients table with the ingredients that need to be purchased to satisfy a given shopping list's date coverage. This is accomplished by checking all of the ingredients in a user's Ingredients table and calculating what their total quantity would be by the end date of the created shopping list. If this quantity would be less than the threshold quantity specified for an ingredient, the function calculates the quantity of an ingredient that would need to be purchased for the total quantity of the ingredient to be above its threshold quantity by the end date of the shopping list. This quantity is then inserted into the Shopping List Ingredients table with the appropriate ingredient, shopping list, and user IDs.

Finally, ing_audit_on_shop_list triggers before a row is deleted from Shopping Lists and executes the ing_audit_on_shop_list() function. This function serves to insert into Ingredient Audits the appropriate audit logs specifying that ingredients that were previously set to be purchased and added to a user's ingredient quantities will now no longer be purchased. This is only done when the shopping list being deleted has a start date in the future, as it is assumed that shopping lists in the past have been purchased and their ingredients added to a user's kitchen. This trigger and function exist with relation to the Shopping List table instead of the Shopping List Ingredients table because entries into Ingredients Audit require a date and time to be specified for when any given audit log is set to occur. This information would have already been lost with the deletion of the Shopping Lists table entry in question, and thus any trigger on Shopping List Ingredients would have to use contrived methods to recover the dates in question.

Shopping List Ingredients

The Shopping List Ingredients table contains all of the Shopping List - Ingredient pairings required to satisfy a user's ingredient requirements for the time period specified by any given shopping list. In this case, satisfaction is defined as ensuring that after all of the meals specified in the shopping list's coverage range have been created, a user's ingredients will all have quantities greater than their threshold quantity. The data associated with any given Shopping List Ingredient entry consists of the ingredient and recipe IDs that pair the necessary ingredients to the appropriate shopping lists, the quantity of a given ingredient that must be purchased (which is a whole-number multiple of the quantity of an ingredient that a user can purchase, as defined by the number of servings that one receives when purchasing an ingredient and the quantity of ingredient in a serving), and the user ID associated with a given recipe-ingredient pairing.

The only trigger associated with Shopping List Ingredients is ing_audit_on_shop_list_ing, which executes the function ing_audit_on_shop_list_ing() before rows are inserted, updated, or deleted in Shopping List Ingredients. This function serves to log in Ingredients Audit every instance where a shopping list ingredient adds ingredients back to a user's "pantry" (which occurs when shopping list ingredients are inserted and when they are updated with a larger value of ingredient to purchase) and when they would reduce the quantity of ingredient being added (which occurs when shopping list ingredients are deleted or updated with a smaller ingredient quantity).

Ingredient Audits

The Ingredient Audits table tracks all of the instances wherein a user uses any quantity of any ingredient as a result of their assignment of meals and shopping lists. The information used to track this consists of the ID of any given audit in the table, the ingredient ID of the ingredient being audited, the recipe ID of which recipe was associated with a particular usage of an ingredient, the meal ID of which meal was associated with a given ingredient's use, the shopping list associated with any given purchase of an ingredient, the table whose insert, update, or deletion prompted the creation of an audit log, the table operation that caused the creation of the audit log, and the associated user ID.

The only trigger on the Ingredients Audit table is shop_list_ing_on_ing_aud, which fires the function shop_list_ing_on_ing_aud() after a row is inserted. This function serves to automatically update the quantities specified by entries in the Shopping List Ingredients table in response to the ingredient quantities changing on given dates, which are represented in the inserted audit logs. For example, if a user has a shopping list set two weeks from now and inserts a new meal with some number of recipes between now and the end date of that shopping list, the shopping list must be updated with the appropriate quantities of ingredients that need to be purchased should that meal preparation cause the associated ingredient quantities to fall below their threshold. Likewise, if a user creates a shopping list that would span the week before anothing shopping list, the purchase of ingredients specified by that shopping list may cause some ingredients to no longer need to be purchased in the following shopping list. Thus, when a row is inserted into the Ingredients Audit table, the quantity of ingredient that needs to be purchased for a given shopping list is recalculated. This recalculation occurs for all shopping lists that follow the date specified by an inserted row, with the calculation occurring in chronological order. In the event that a calculation demonstrates that a shopping list ingredient must be inserted, updated with a different quantity, or deleted, the current recalculation stops. This is because the addition of a new entry into Shopping List Ingredients will be accompanied with a new Ingredient Audits insertion, which will then trigger shop_list_ing_on_ing_aud and fire shop_list_ing_on_ing_aud() again, albeit starting from a later date.

Database Design

The database was designed with the intent of maintaining third normal form, which is a database schema design wherein the goal is to reduce data anomalies. This is accomplished by having all attributes be functionally dependent on the primary key of a table. As a rule, null entries are not permitted in any of the tables seen in Figure 1, aside from the exception of foreign key values in Ingredients Audit. This was done to avoid issues related to the input of data with default values of null, which would lock up the functioning of the database.

Component List

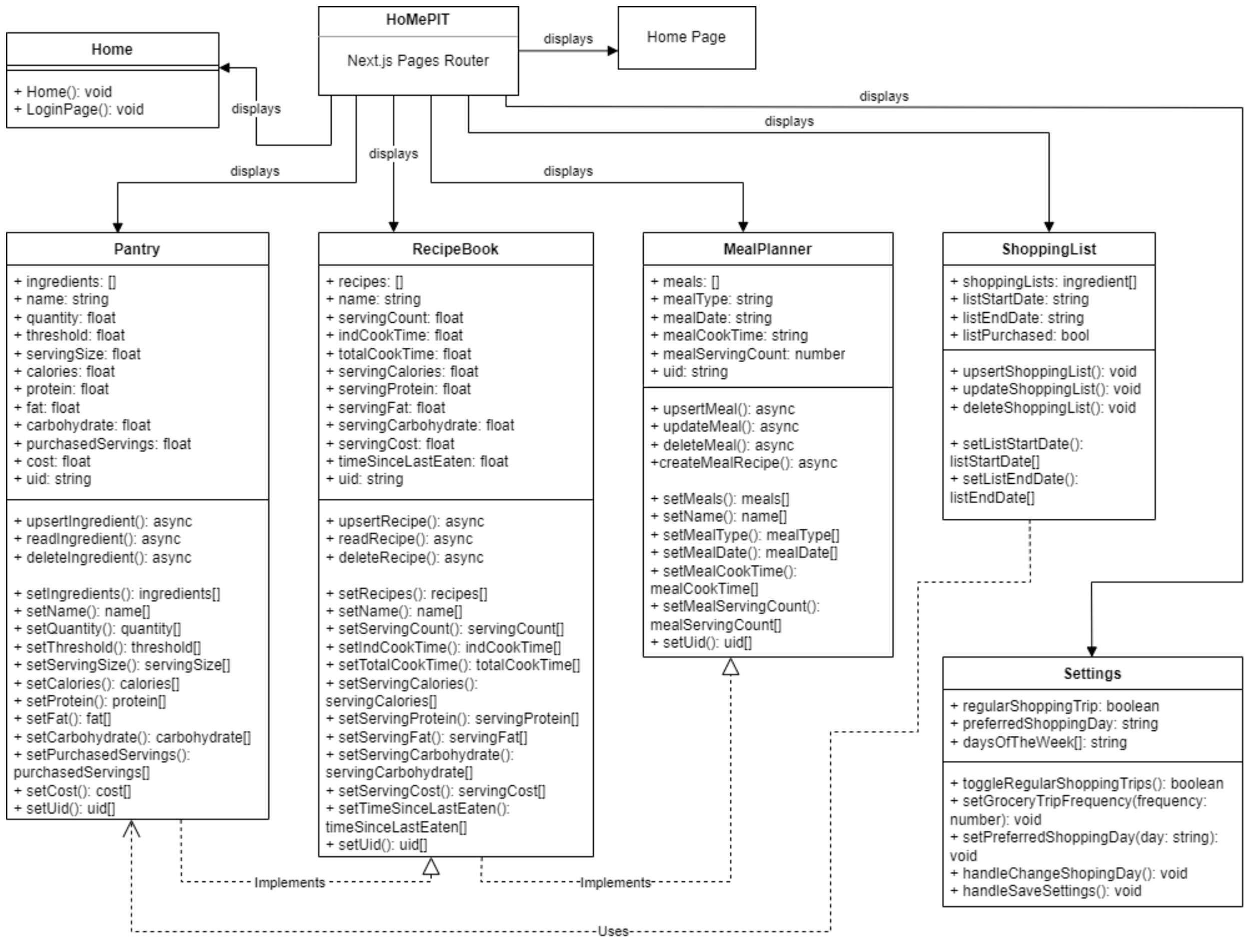
HoMePIT
Next.js Pages Router
displays
Home Page
Home
+ Home(): void
+ LoginPage(): void
displays
displays
displays
displays
displays
displays
Pantry
+ ingredients: []
+ name: string
+ quantity: float
+ threshold: float
+ servingSize: float
+ calories: float
+ protein: float
+ fat: float
+ carbohydrate: float
+ purchasedServings: float
+ cost: float
+ uid: string
+ upsertIngredient(): async
+ readIngredient(): async
+ deleteIngredient(): async
+ setIngredients(): ingredients[]
+ setName(): name[]
+ setQuantity(): quantity[]
+ setThreshold(): threshold[]
+ setServingSize(): servingSize[]
+ setCalories(): calories[]
+ setProtein(): protein[]
+ setFat(): fat[]
+ setCarbohydrate(): carbohydrate[]
+ setPurchasedServings(): purchasedServings[]
+ setCost(): cost[]
+ setUid(): uid[]
RecipeBook
+ recipes: []
+ name: string
+ servingCount: float
+ indCookTime: float
+ totalCookTime: float
+ servingCalories: float
+ servingProtein: float
+ servingFat: float
+ servingCarbohydrate: float
+ servingCost: float
+ timeSinceLastEaten: float
+ uid: string
+ upsertRecipe(): async
+ readRecipe(): async
+ deleteRecipe(): async
+ setRecipes(): recipes[]
+ setName(): name[]
+ setServingCount(): servingCount[]
+ setIndCookTime(): indCookTime[]
+ setTotalCookTime(): totalCookTime[]
+ setServingCalories(): servingCalories[]
+ setServingProtein(): servingProtein[]
+ setServingFat(): servingFat[]
+ setServingCarbohydrate(): servingCarbohydrate[]
+ setServingCost(): servingCost[]
+ setTimeSinceLastEaten(): timeSinceLastEaten[]
+ setUid(): uid[]
MealPlanner
+ meals: []
+ mealType: string
+ mealDate: string
+ mealCookTime: string
+ mealServingCount: number
+ uid: string
+ upsertMeal(): async
+ updateMeal(): async
+ deleteMeal(): async
+createMealRecipe(): async
+ setMeals(): meals[]
+ setName(): name[]
+ setMealType(): mealType[]
+ setMealDate(): mealDate[]
+ setMealCookTime(): mealCookTime[]
+ setMealServingCount(): mealServingCount[]
+ setUid(): uid[]
ShoppingList
+ shoppingLists: ingredient[]
+ listStartDate: string
+ listEndDate: string
+ listPurchased: bool
+ upsertShoppingList(): void
+ updateShoppingList(): void
+ deleteShoppingList(): void
+ setListStartDate(): listStartDate[]
+ setListEndDate(): listEndDate[]
Settings
+ regularShoppingTrip: boolean
+ preferredShoppingDay: string
+ daysOfTheWeek[]: string
+ toggleRegularShoppingTrips(): boolean
+ setGroceryTripFrequency(frequency: number): void
+ setPreferredShoppingDay(day: string): void
+ handleChangeShopingDay(): void
+ handleSaveSettings(): void
Implements
Implements
Uses

Figure 2. The final Component Diagram of HoMePIT v1.

Navbar and Sidebar

Homeput uses Material UI libraries to create a responsive navigation bar at the top of the page, and a side navigation bar using MUI's <Drawer/> component. This was opted for over a traditional navigation bar due to a sidebar being more mobile-friendly, something which was deemed important by the development team. Upon clicking the three-lined icon in the top left of the screen, the sidebar will appear, displaying the various tabs available to the user. These pages will be discussed in further detail, with the primary ones being the Homepage, Pantry, Recipe Book, Meal Planner, and Shopping List. The Settings page is intended to play a minor role for shopping list generation, and the Profile page exists on the sidebar but has not yet been implemented.

Home Page and Authentication

The first major component of HoMePIT that the user is introduced to is the Home page, whose primary purpose is to handle user sign in and sign out. Upon loading the page the user will be presented with HoMePIT's logo and a request to sign in to begin using the application. Upon clicking the "Login" button, the user will be taken to a page where they will be able to sign in with their Google account, all of which is handled through Supabase's integration with Google OAuth. Following a successful sign in, the user will be redirected back to HoMePIT's Homepage, where they will have the option to logout.

The Home page itself operates through the use of two functions, Home(), which renders the "Logout" page if there is no active session, and LoginPage(), which renders the initial sign in page as described above. The LoginPage contains the async function loginWithGoogle(), which uses the supabase client to redirect the user to the Google sign in page. The redirect URL for this process is defined within the Authentication page of the teams Supabase backend.

Supabase Client

The frontend of HoMePIT interacts with the Supabase database through the Supabase Client, a third party library offered by Supabase. This library handles the .env file in the project, which contains both the anon key and connection string, designated as NEXT_PUBLIC_SUPABASE_URL and NEXT_PUBLIC_SUPABASE_ANON_KEY, respectively. The supabase client is used in nearly every component of HoMePIT, as it is needed to properly create and update necessary data.

Firebase Hosting

HoMePIT uses Google Firebase as its hosting service – a shift from earlier notions of using Replit. Initially, when the team was developing on Replit it was soon decided to shift to using VSCode as the primary IDE, as the team found Replit to be very tedious and unfriendly considering the project's scope. Despite the shift, the team intended to continue to use Replit to function as the server for the project, however after having difficulty with implementation as HoMePIT grew in scale it became clear that Replit would not be an optimal choice in the long run. Soon afterward, firebase hosting was implemented. Initially the team was using the free Spark plan, but as development progressed upgrading to the Blaze plan was required.

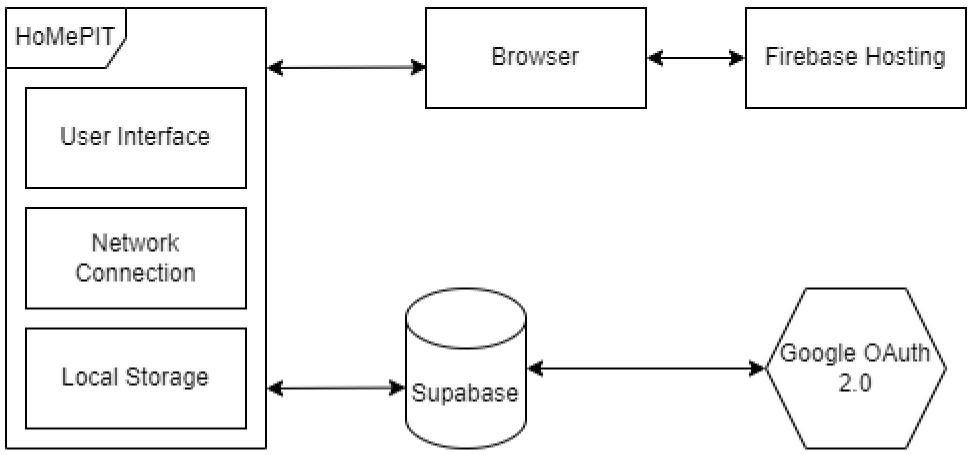

Figure 3. The final Block Diagram of HoMePIT v1.

Pantry

The Pantry page is the most foundational component of HoMePIT, as it handles all information regarding ingredients, their associated data, and their display to the user. On the surface, the Pantry page initially consists of an empty table, with a button labeled "Add New Ingredient" that upon clicking will open a modal which the user can use to create new ingredients to add to their Pantry. The data that the user can add consists of the ingredient name, quantity, threshold (the minimum amount needed in the Pantry), serving size, calories, protein, fat, carbohydrate, purchased servings, and purchase cost. The user may then click the "Submit" button, which will create a new ingredient that is associated with the user's unique identifier that was created upon signing in. From here, the created ingredient will appear in the table and the user will have options to either edit or delete the desired ingredient. Upon clicking the edit icon, the same modal as described previously will appear and the user will be able to enter the new information regarding the selected ingredient. Upon submission, the ingredient will display its new state to the user in the table. Furthermore, upon clicking the delete icon, the selected ingredient will be removed from both the table and the database.

The code structure of the Pantry page, and all pages for that matter, can be broken into three main parts, with the first being required import statements, the second being the primary function(s), and the third being the return statements of said function(s). Being a Next.js project, HoMePIT inherits much of its structure from React, with the previously described code layout being a good example of that. Much of the import statements are related to third-party tools, which will be discussed later. As for the main function, Pantry(), it is divided into three sections with the beginning of the function consisting of useState() variables, in which the various components of what constitute an ingredient are defined to be used later in the program. Two such examples of this are ingredients and its setter setIngredients, which defines and sets the ingredients list, and name with its setter setName, which defines and sets the name attribute as a string. The rest of the state variables are defined as type "any," although within the database they are defined as definite types. This discrepancy will be touched on later in the Future Work section.

Following the state variables comes the core logic for the creation, editing, reading, and deletion of ingredients from the database. Initially the creation and updating were handled through separate async functions, createIngredient and updateIngredient , however these were removed in favor of upsertIngredient, which served to simplify the code and provide a single method that handles the user's needs. In contrast to .insert or .update, performing .upsert on a table in the database will update a given row based off of a key if it already exists, or create a new row if it does not. In this case, the ing_id of an ingredient serves as that key. This works well for the needs of HoMePIT and allows for only needing a single modal for adding or editing a given ingredient, in which only a single method needs to be called. Beyond upsertIngredient, the methods readIngredient and deleteIngredient also exist, the former of which is used to display to the user their ingredients in the table, and the latter which removes the ingredient from the table and database entirely. Beyond these methods, this section of the Pantry's code also contains the method getUser which initializes the user_id that will be assigned to every ingredient created by the user. This ensures that the user only has access to ingredients created by them.

Lastly, the return statement for the Pantry contains all of the HTML and Material UI code needed to provide a visual display to the user. This section structures the ingredient table through various MUI components such as<TableContainer/>, <Table/>, <TableHead/>, and <TableRow/>

Additionally, many native HTML elements such as <button/> and <select/> were overridden by their MUI counterparts, such as <Button/> and <Select/>. Using MUI over HTML often provided a much better look and feel for the application than that provided by native HTML.

Recipe Book

Following the Pantry, the next page available to the user is the Recipe Book, which builds off of the Pantry in that whenever a new recipe is created, ingredients created previously in the Pantry can be added to it, creating new "Recipe Ingredients," or data objects which by associating a given ing_id with a rec_id, create an association between the two. To the user, the Recipe Book visually is very similar to the Pantry in that it consists of a table that displays data about a data object – recipes in this case. There is an "Add New Recipe" button at the top of the page that, when clicked, opens a modal where the user can input recipe data as well as add ingredients to the soon to be created recipe.

As stated, recipes are similar to ingredients in that they are made up of various attributes – in this case those being recipe ID, recipe name, serving count, cook time, serving calories, serving protein, serving fat, serving carbohydrate, serving cost, time since last eaten, and user ID. Just as in the Pantry these are set as state variables in the beginning of the RecipeBook() function to be used later on in the program. Certain attributes, such as everything encompassing the macronutrient and cost data, are not input directly by the user but are calculated in the backend by attaining the sum of a given column of Recipe Ingredients. For instance, if the user were to create an "omlette" recipe, and it consisted of 2 eggs, 1 small onion, and various other ingredients, that omlette's total calorie count per serving would be the sum of all ingredient calories divided by the number of servings of that recipe. These calculations aree handled in the backend, and the only influence the user has over this process is the data they insert for each ingredient, as well as the selected ingredients used in the creation of a new recipe.

Just as in the Pantry, the Recipe Book returns HTML and MUI code to provide the user with a display of their recipes. This display is nearly identical to that found in the Pantry, with the only notable differences being the data presented to the user. Within each row on the table exists a button labeled "Nutrition," which when clicked will present the user with a modal that displays relevant nutrition information such as total calories, total protein, total fat, total carbohydrates, etc. This data is calculated in the backend and assigned to the row in question upon thickening this button. Beyond this feature, the user once again has the option of editing or deleting the selected recipe.

Meal Planner

The next page accessible to the user is the Meal Planner, which contrary to the previous two pages does not consist of a table to display data. Rather, this page contains a calendar which displays the days of the current month, with options to view previous or future months, weeks, and days. Immediately to the left of this is a column where individual meals created by the user appear when created. Just as with the previous pages, Meals are created via a button, in this case labeled "Add Meal." Upon clicking here a modal will appear with options to add meal types from a dropdown, date and time of cooking, time to cook, serving count, and a dropdown containing a list of all recipes created by the user. Additionally, the meal types specified here are Breakfast, Brunch, Lunch, Dinner, Dessert, and Snack. Upon submission, recipes included under a given meal type will in turn create a new Meal Recipe, which functions similarly to how

Recipe Ingredients do in the Recipe Book. Essentially, Meal Recipes are created by assigning each rec_id to the given meal_id, with further information such as nutrition data being calculated in the backend. Much of the desired functionality for the Meal Planner did not go into effect and will be discussed later in this document.

Shopping List

The Shopping list page of HoMePIT is intended to be generated automatically based on whether or not an ingredient is approaching or is under its defined threshold quantity, something set by the user in the Pantry page. This page is intended to serve as a reference to the user for the ingredients they need to buy during their next shopping trip. As it stands, the Shopping List lacks much of what it was intended to be – something that will be discussed later on in this paper. As it is currently, this page displays only a list of items that have a quantity value of less than or equal to their threshold value. There is no current functionality relating to date ranges, cost calculation, wtc as stated in previous documentation.

Settings

The settings page is intended to serve as a location for the user to toggle on or off the "Regular shopping trips?" feature, which will define whether or not shopping lists are generated at regular intervals. It possesses the toggle switch, an input field with a dropdown to select a day of the week to generate shopping lists on, and a "Save Settings" button intended to save the settings selected by the user. This page's code is very simple, and consists This page lacks any functionality for the present.

How to Deploy

In order to simply use HoMePIT, the user needs only to visit the link provided in the GitHub repository to begin using the application. In order to run HoMePIT on one's own machine, the steps can be summarized as follows:

1. Ensure that VSCode is installed and up to date.
2. Navigate to a terminal. Ensure that Git, node.js, and all other tech specified previously in this document is installed and up to date on the machine.
3. Create a directory where the application will be stored.
4. Run "git clone https://github.com/mfugette/HoMePIT-v2.git" to clone the repo.
5. Navigate to the newly created project from the terminal.
6. Run "code ." to open the project in VSCode.
7. Add the .env file to the root folder of the project.
8. Upon opening the code editor, use Ctrl + ` to open a terminal.
9. Run "npm install" to install and update all node packages needed.
10. Run "npm run dev" to start the development server.

From here, the developer will be able to edit and see their changes in real time. These steps were followed by the development team in order to get fresh copies of HoMePIT whenever there was an issue that required a complete reinstall.

Known Bugs and Issues

Database Bugs and Issues

- To prevent users from changing the date range covered by any given shopping list (which was done to simplify shopping list database functionality), any attempt to update a shopping list returns null. However, this still reports a successful updating of a shopping list, even if no information changes. Instead, an error should be reported to reduce the chances of misinterpretation.

Frontend Bugs and Issues

- Issues with displaying nutrition data for ingredients
- Inserting Meals into calendar causes crash
- Duplicate Meals are generated occasionally
- Most ingredient and recipe attributes are forgotten upon edit
- UI sometimes will stretch off of the screen when screen resolution is insufficient.

Future Work

The development of HoMePIT's functionalities in code occurred over the course of the latter half of the fall semester. This was a span of time roughly two months in length that coincided with the increased workload associated with the end of the semester. Future work to aspire to in an update to HoMePIT has been selected based on the assumption that approximately twice as much work could be completed in an additional span of code-development that is twice the length of the span that was experienced. Furthermore, it is assumed that the development of future work would be aided by the fact that the current developers are now much more familiar with the functioning of HoMePIT than they were when they started development. Thus, it is assumed that this would allow for even more work to be effectively completed. However, even in the case that development were passed on to another group of developers, it can be assumed that they would match and exceed the current developers' familiarity within two months, which would result in another two months of equally and more effective development.

Backend Features

- More permissive shopping list insertion, such that a user can input a shopping list spanning part or all of one or more shopping lists and subsume those shopping lists into itself.
- Have shopping lists confirmed by the user rather than assumed to have been purchased.
- Allow shopping lists to have their ranges updated.
- Have shopping list ingredients update in response to updates to ing_threshold_qnt, ing_purchase_serving_cnt, ing_serving_size in Ingredients.
- Add purchase location functionality, such that a user can specify the purchase location(s) of an ingredient and have that location specified in a shopping list.
- Add purchase location optimization for shopping lists, such that an optimal or near-optimal set of purchase locations are selected to minimize the number of locations that need to be visited for any given shopping list.
    - This requires the use of heuristics, as the optimization of shopping list locations is a variant of the minimal-hitting-set problem, which is NP-hard.
- Add leftover functionality, such that a user can specify that a meal will have more servings prepared than consumed and track the existence and expiry of those leftovers.
- Add expiring ingredients functionality, such that a user can specify how long ingredients remain fresh and have ingredients automatically added to a shopping list if they would expire.
- Better compatibility with users inputting data for the past, whether that be with respect to meals, shopping lists, or other data that HoMePIT handles.

Frontend Features

- Purchase location UI
- Leftovers UI
- Ingredient expiration UI
- UI for confirming that a shopping list has been purchased
- Real Time Database subscription functionality, enabling tables to update with correct information as necessary
- Improved Frontend visuals
- Table sorting functionality, UI, and UX
- Ingredient and recipe tag UI and UX
- User ability to input non g or mL units as desired
- Improved compatibility with Android
- Settings page functionality, UI, and UX
- Automatically shopping list creation out a user-specified certain span of time