

Service Layer Design Document

1. Overview

The Recip-Ease application utilizes RESTful api to allow the web pages to communicate with the database. Frankly it is about the simplest way to build the UI/database infrastructure, and it was simple to implement the necessary CRUD operations required. Also, with JSON responses, the data is structured on return from the database and easy to handle programmatically. The set up is highly scalable, though in the case of this project that is of less consequence. RESTful processes do mesh well with MongoDBs document-based data model.

Database structure and endpoints were easily adaptable to the needs of Recip-Ease. Javascript is simple to develop, and flexible enough to handle the needs of the project. As can be seen from the screenshots, the database and the RESTful api are fairly well developed. While we are early in the Capstone project lifecycle, I would remind you that I have had this project in mind since about 2020. My vision is solid, and I have done a lot of research on how it would work, and how I would make it work functionally, have been researched and settled a long time ago.

For the purpose of this document, we will progress through the workflow indicating which endpoints come into play. Graphical examples of the appropriate workflows will appear at the end of the discussion of each portion of the website. Some CRUD operations like DELETE for users and some others like PUT for the ingredients collections have been back-burnered as they are not necessary for the functionality of the MVP.

2. Service Endpoints Summary

HTTP Method	Endpoint	Page(s)	Purpose
POST	/api/users/register	User Login/Create Account Page	Register a new user
POST	/api/users/login	User Login/Create Account Page	Authenticate a user
PUT	/api/users/:id	User Login/Create Account Page	Update user email or password
GET	/api/users/:id	User Login/Create Account Page	Retrieve user details
POST	/api/recipes/basicinfo	Recipe Entry Page	Create a new recipe
GET	/api/recipes	Recipe Entry Page Recipe Search Page	Retrieve all recipes for the logged-in user
GET	/api/recipes/:id	Home Page Recipe Entry Page Recipe Search Page	Retrieve a specific recipe by ID
PUT	/api/recipes/:id	Recipe Entry Page	Update an existing recipe
DELETE	/api/recipes/:id	Recipe Entry Page Recipe Search Page	Delete a recipe
GET	/api/recipes/random/4	Home Page	Retrieve 4 random recipes for homepage slideshow
PUT	/api/recipes/:id/ingredients	Recipe Entry Page	Add/update an ingredient to a recipe

HTTP Method	Endpoint	Page(s)	Purpose
PUT	/api/recipes/:id/instructions	Recipe Entry Page	Add/update recipe instructions
DELETE	/api/recipes/:id	Recipe Entry Page Recipe Search Page	Delete a specific recipe
DELETE	/api/recipes/:id/instructions	Recipe Entry Page	Delete an instruction from a recipe
DELETE	/api/recipes/:id/ingredients	Recipe Entry Page	Delete an ingredient from a recipe
GET	/api/:collectionName	Recipe Entry Page	Retrieve common ingredients for type of recipe (like “cake” in the dessert_ingredient collection)
PUT	/api/:collection/:ingredient/increment	Recipe Entry Page	When an ingredient is added to a recipe, the “usage_count” is incremented to increase the ingredient’s popularity for later lists

3. Endpoint Descriptions & Example Requests

User Registration

Note: While I believe the website and functionality have been well thought out, as development proceeds, no doubt further functionality and endpoints will likely be added. However, I believe the current state of the website and these endpoints will satisfy the requirements of the MVP. One area which would concern me greatly were this website made publicly accessible, would be security. Currently, the API is open and public-facing without authentication and authorization. If I should plan to take this project beyond the MVP, security would be my first implementation. Being a non-SQL database, I at least don’t have to worry about injection attacks so much, though I understand MongoDB isn’t impervious to such things.

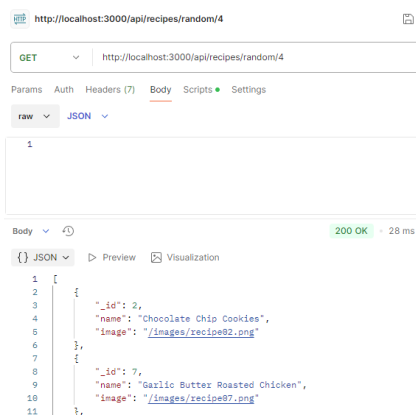
Home Page Functionality

Other than navigating to other pages, the home page doesn’t have much to do with our persona’s workflow. A couple things I will be implementing are the photo display at the top, and the “Featured Recipes” at the bottom. Both utilize the same endpoint, GET /api/recipes/random/4, which selects four of the recipes at random from the database and displays photos and descriptions of the recipes.

Retrieve Random Recipes for Homepage

Endpoint: GET /api/recipes/random/4

Example Request:



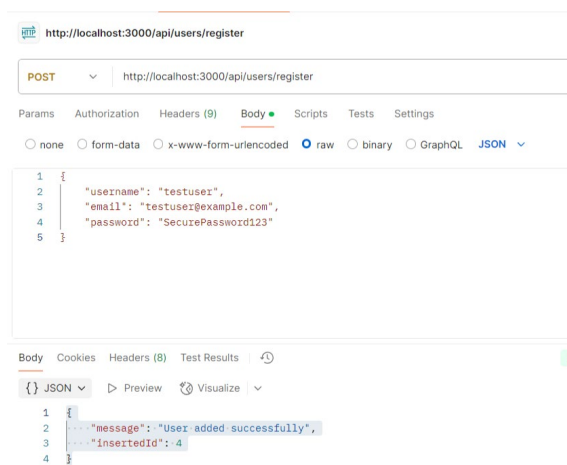
Error Response (500 Internal Server Error):

```
{
  "error": "Database error while fetching random recipes"
}
```

The most common navigation workflow for most users will be to navigate to the User Sign-In/Create User page. Recipes can be identified by the user creating them, and the user can list and view their recipes. Custom cookbooks, or collections of recipes will be a stretch feature. Currently, when a user either registers or logs in, the endpoint returns a success response along with the user's `_id`, which can be stored in a variable for including in new recipe documents or for searches. The first endpoint is for registering a new user. It accepts a username, email, and password, which it hashes when storing. The API also checks for duplicate usernames or emails.

New User Registration Endpoint: POST /api/users/register

Example Request/Response:



http://localhost:3000/api/users/register

POST http://localhost:3000/api/users/register

Params Authorization Headers (9) Body Scripts Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL JSON

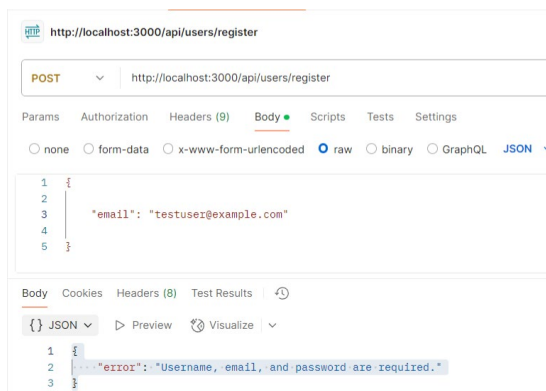
```
1 {
2   "username": "testuser",
3   "email": "testuser@example.com",
4   "password": "SecurePassword123"
5 }
```

Body Cookies Headers (8) Test Results

JSON Preview Visualize

```
1 {
2   "message": "User added successfully",
3   "insertedId": 4
4 }
```

Error Response (400 Bad Request - Duplicate User):



http://localhost:3000/api/users/register

POST http://localhost:3000/api/users/register

Params Authorization Headers (9) Body Scripts Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL JSON

```
1 {
2   "email": "testuser@example.com"
3 }
```

Body Cookies Headers (8) Test Results

JSON Preview Visualize

```
1 {
2   "error": "Username, email, and password are required."
3 }
```

Once a user account has been created, users can log in with their username and password. The API returns the user's document `_id`.

User Login

Endpoint: POST /api/users/login

Example Request:

HTTP <http://localhost:3000/api/users/login>

POST <http://localhost:3000/api/users/login>

Params Authorization Headers (9) Body Scripts Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL [JSON](#)

```
1 {
2   "username": "testuser",
3   "password": "SecurePassword123"
4 }
5
```

Body Cookies Headers (8) Test Results [↻](#)

[{} JSON](#) [▶ Preview](#) [🔗 Visualize](#)

```
1 {
2   "message": "Login successful",
3   "userId": 4
4 }
```

Error Response (401 Unauthorized):

HTTP <http://localhost:3000/api/users/login>

POST <http://localhost:3000/api/users/login>

Params Authorization Headers (9) Body Scripts Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL [JSON](#)

```
1 {
2   "username": "testuser",
3   "password": "Not right"
4 }
5
```

Body Cookies Headers (8) Test Results [↻](#)

[{} JSON](#) [▶ Preview](#) [🔗 Visualize](#)

```
1 {
2   "error": "Invalid username or password."
3 }
```

Error Response (400 Bad Request):

HTTP <http://localhost:3000/api/users/login>

POST <http://localhost:3000/api/users/login>

Params Authorization Headers (9) Body Scripts Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL [JSON](#)

```
1 {
2   "username": "testuser@example.com",
3   "password": "SecurePassword123"
4 }
5
```

Body Cookies Headers (8) Test Results [↻](#)

[{} JSON](#) [▶ Preview](#) [🔗 Visualize](#)

```
1 {
2   "error": "username and password are required."
3 }
```

Users will have the ability to change either their email address or password. The first step in that process is to retrieve the user details. Once logged in and the `userId` stored, the system retrieves the user's information with a GET endpoint for the user, then they can change either their email address or password.

Retrieve User Details

Endpoint: GET `/api/users/:id`

Example Request (Successful):

http://localhost:3000/api/users/4

GET http://localhost:3000/api/users/4

Params Authorization Headers (7) **Body** Scripts Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL **JSON** ▾

1

Body Cookies Headers (8) Test Results | ↻

{ } JSON ▾ ▶ Preview 🌐 Visualize ▾

```
1 {
2   "_id": 4,
3   "username": "testuser",
4   "email": "testuser@example.com",
5   "created_at": "2025-03-20T17:05:04.785Z",
6   "updated_at": "2025-03-20T17:05:04.785Z"
7 }
```

Error Response (404 Not Found):

http://localhost:3000/api/users/6

GET http://localhost:3000/api/users/6

Params Authorization Headers (7) **Body** Scripts Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL **JSON** ▾

1

Body Cookies Headers (8) Test Results | ↻

{ } JSON ▾ ▶ Preview 🌐 Visualize ▾

```
1 {
2   "error": "User not found."
3 }
```

Once the user details are displayed, the user can update their email or password.

Update User Details (change email or password)

Endpoint: PUT /api/users/:id

Example Request:

http://localhost:3000/api/users/4

PUT http://localhost:3000/api/users/4

Params Auth Headers (9) **Body** Scripts Tests Settings

raw ▾ **JSON** ▾

```
1 {
2   "email": "newemail@example.com"
3 }
```

Body ▾ ↻ **200 OK** ▾

{ } JSON ▾ ▶ Preview 🌐 Visualize ▾

```
1 {
2   "message": "User updated successfully",
3   "modifiedCount": 1
4 }
```

Database record updated:

```
_id: 4
username : "testuser"
email : "newemail@example.com"
password_hash : "$2b$10$f3qRpieXegLGce4X/PLdyeylX9Eu7gLAYyx9qH7TDy2Hza48qD0Ii"
created_at : "2025-03-20T17:05:04.785Z"
updated_at : "2025-03-20T17:21:17.154Z"
```

Error Response (404 Not Found):

The screenshot shows a REST client interface with the URL `http://localhost:3000/api/users/6`. The method is set to `PUT`. The request body is a JSON object: `{ "email": "newemail@example.com" }`. The response status is `404 Not Found`. The response body is a JSON object: `{ "error": "User not found." }`.

Once logged in, most persona will likely go to the recipe Search page. There, they will have the several options for searching for and displaying their and others, recipes. There are several endpoints for these functions and others on the page; the first is to retrieve all recipes in the database, the second is to retrieve just the user's recipes, and the last is to retrieve recipes using either category and type or with a search term. The same endpoint works for either of the last options.

Retrieve All Recipes Endpoint: GET /api/recipes

Example Request:

The screenshot shows a REST client interface with the URL `http://localhost:3000/api/recipes`. The method is set to `GET`. The response status is `200 OK` with a response time of `9 ms`. The response body is a JSON array containing one recipe object: `[{ "_id": 1, "user_id": 1, "name": "Chocolate Cake", "category": "Desserts", "type": "Cake", "ingredients": { "All-purpose flour": { "quantity": 2.5, "unit": "Cup" }, "Granulated sugar": { ... } } }]`.

Retrieve Recipe By ID Endpoint: GET /api/recipes/:id

Example Request:

HTTP GET http://localhost:3000/api/recipes/6

Params Auth Headers (7) Body Scripts Settings

raw JSON

1

Body 200 OK • 10 ms • 1.2

```
{
  "_id": 6,
  "user_id": 1,
  "name": "Beef Stroganoff",
  "category": "Main Courses",
  "type": "Beef",
  "ingredients": {
    "Beef sirloin": {
      "quantity": 1,
      "unit": "Lbs."
    },
    "Salt": {
      "quantity": 1,
      "unit": "TSP"
    },
    "Black pepper": {
      "quantity": 0.5
    }
  }
}
```

Error Response (404 Not Found):

HTTP GET http://localhost:3000/api/recipes/12

Params Auth Headers (7) Body Scripts Settings

raw JSON

1

Body 404 Not Found • 0.1 ms • 0.0

```
{
  "error": "recipe not found."
}
```

Recipe Search Endpoint: GET /api/recipes/search?query=<search_term>

Example Request:

HTTP GET http://localhost:3000/api/recipes/search?query=chocolate

Params Auth Headers (7) Body Scripts Settings

raw JSON

1

Body 200 OK • 9 ms • 0.0

```
[
  {
    "_id": 1,
    "user_id": 1,
    "name": "Chocolate Cake",
    "category": "Desserts",
    "type": "Cake",
    "ingredients": {
      "All-purpose flour": {
        "quantity": 2.5,
        "unit": "Cup"
      },
      "Granulated sugar": {
        "quantity": 1.5,
        "unit": "Cup"
      }
    }
  }
]
```

Error Response (404 Not Found):

The screenshot shows a REST client interface with the URL `http://localhost:3000/api/recipes/search?query=absinth`. The method is `GET`. The response status is `404 Not Found`. The response body is displayed in JSON format:

```
1 {
2   "error": "No recipes found matching your search."
3 }
```

The final set of endpoints as far as the MVP is concerned is the creation, modification, and deletion of recipes by the user. To recap, to create a recipe, the user will enter some details like recipe name, as well as category and type. I will likely add “description” to this basic information in the MVP as well. Initially, given the workflow of the recipe entry page, the recipe document will be created at this point, the `recipe_id` stored, and as the user enters the ingredients and instructions, a PUT endpoint will add them to the recipe document. The returned “`recipe_id`” field is loaded into a variable and utilized for later PUT endpoints to add ingredients and instructions to the recipe.

Create New Recipe Endpoint: POST /api/recipes/basicinfo

Example Request:

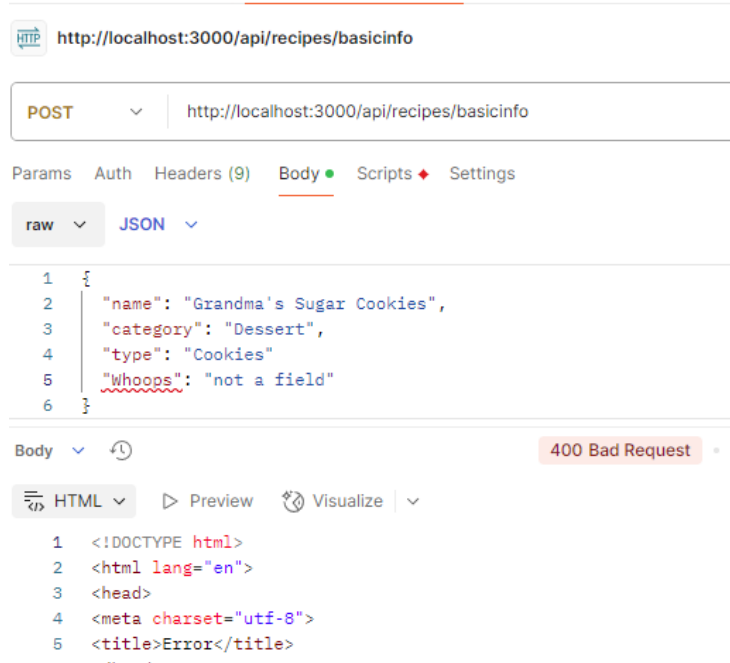
The screenshot shows a REST client interface with the URL `http://localhost:3000/api/recipes/basicinfo`. The method is `POST`. The response status is `201 Created`. The response body is displayed in JSON format:

```
1 {
2   "name": "Grandma's Sugar Cookies",
3   "category": "Dessert",
4   "type": "Cookies"
5 }
```

The response body is also shown in a preview view:

```
1 {
2   "message": "Basic recipe information created successfully",
3   "recipe_id": 10
4 }
```


Malformed Request (400 Bad Request):



```
1 {
2   "name": "Grandma's Sugar Cookies",
3   "category": "Dessert",
4   "type": "Cookies"
5   "Whoops": "not a field"
6 }
```

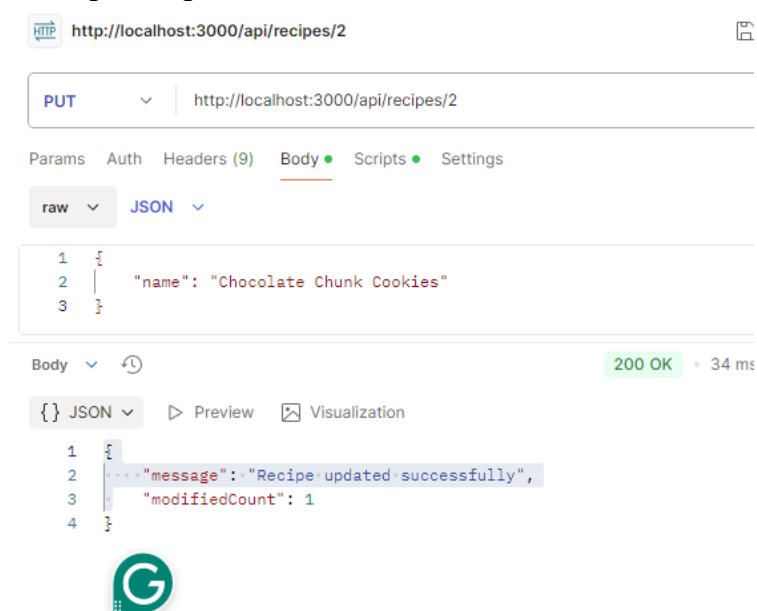
Body 400 Bad Request

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4 <meta charset="utf-8">
5 <title>Error</title>
6 ..
```

The user can update the basic recipe information such as name, category, type. Those are updated with this endpoint. Ingredients and recipe instructions are handled by other endpoints for a variety of reasons, most of which have to do with the workflow design.

Update Recipe Information Endpoint: PUT /api/recipes/:id


Example Request:



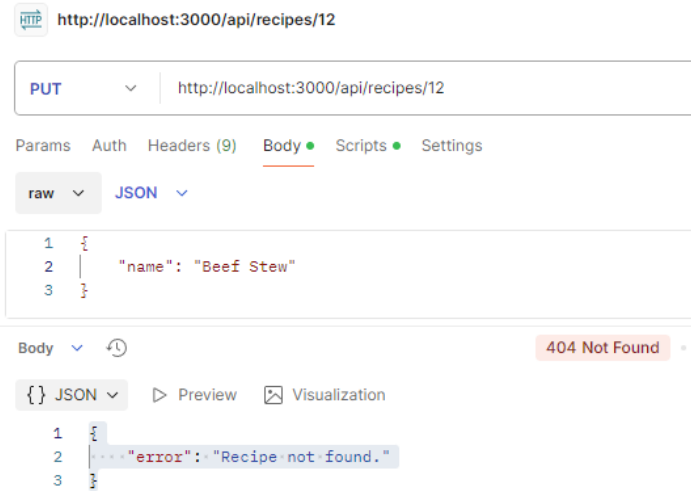
```
1 {
2   "name": "Chocolate Chunk Cookies"
3 }
```

Body 200 OK 34 ms

```
1 {
2   "message": "Recipe updated successfully",
3   "modifiedCount": 1
4 }
```



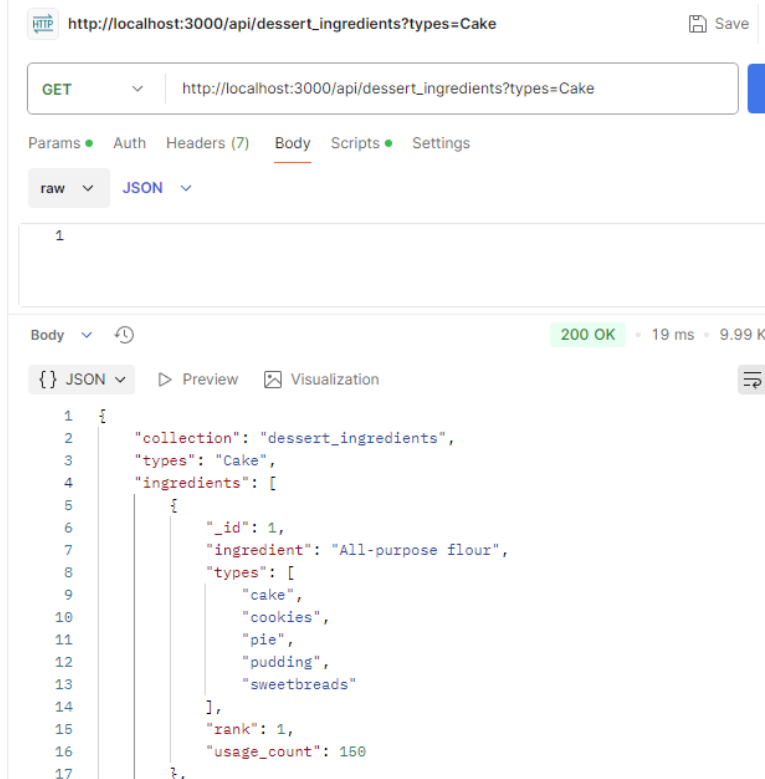
Error Message (404 Not Found):



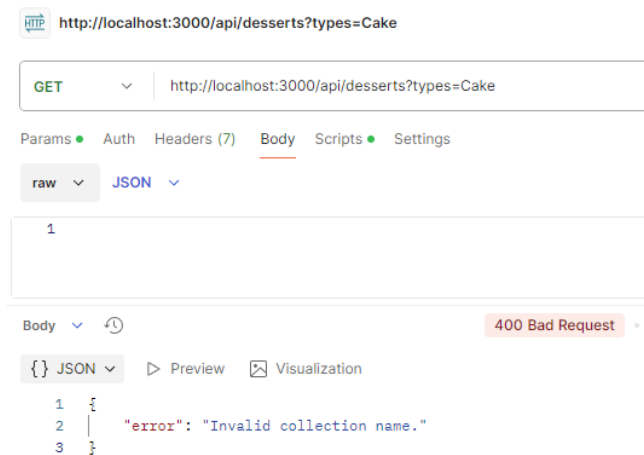
Once the user has entered or updated the basic recipe info, the database will utilize this information to populate a list of the most common ingredients in a scrollable list. The items on the list will be clickable for selection by the user. This endpoint, utilizes the category (which translates into :collectionName) and type to GET the list. (The ingredient collections are preloaded with 100 – 125 common ingredients culled from some 10,000 recipes, and ranked by common use using AI. This is a database administrative function and not a part of the website’s functional capabilities.)

Retrieve Common Ingredients By Type Endpoint: GET /api/:collectionName

Example Request for Ingredients from `dessert_ingredients` for recipe type ‘cake’:



Example Error (400 Bad Request):



HTTP GET http://localhost:3000/api/desserts?types=Cake

Params Auth Headers (7) Body Scripts Settings

raw JSON

1

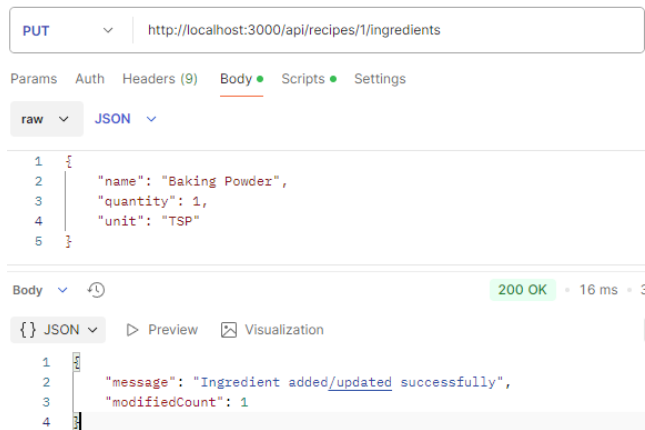
Body 400 Bad Request

```
{
  "error": "Invalid collection name."
}
```

Once the ingredient list is loaded, the user clicks on the ingredient, then click the quantity buttons provided in the left navigation. As they click these, the quantities are loaded into a variable, and when the user hits the “Submit Ingredient” button, a PUT endpoint adds the ingredient to the recipe.

Add or Modify Ingredients to Recipe Document Endpoint: PUT /api/recipes/:id/ingredients

Example Request:



PUT http://localhost:3000/api/recipes/1/ingredients

Params Auth Headers (9) Body Scripts Settings

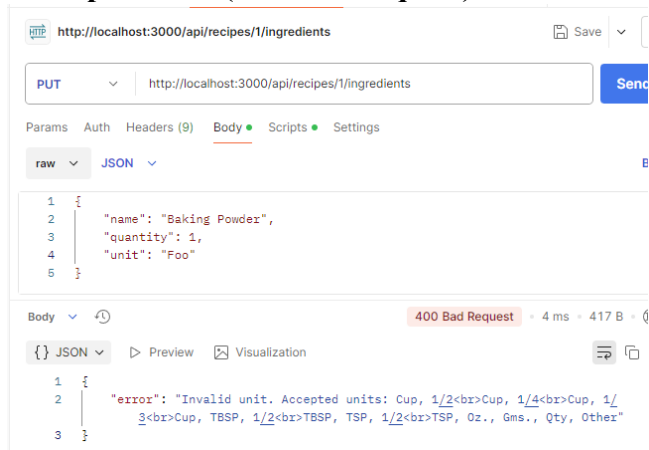
raw JSON

```
{
  "name": "Baking Powder",
  "quantity": 1,
  "unit": "TSP"
}
```

Body 200 OK 16 ms

```
{
  "message": "Ingredient added/updated successfully",
  "modifiedCount": 1
}
```

Example Error (400 Bad Request):



PUT http://localhost:3000/api/recipes/1/ingredients

Params Auth Headers (9) Body Scripts Settings

raw JSON

```
{
  "name": "Baking Powder",
  "quantity": 1,
  "unit": "Foo"
}
```

Body 400 Bad Request 4 ms 417 B

```
{
  "error": "Invalid unit. Accepted units: Cup, 1/2, 1/4, 3/4, TBSP, 1/2, TSP, Oz., Gms., Qty, Other"
}
```

Part of the unique functionality of the website is that it essentially “learns” from the frequency that users use individual ingredients in their recipes. Rather than implementing a “Favorites” functionality, the ingredients documents in those collections keep track of a number called “usage_count”. When the ingredients are retrieved from the database they are displayed sorted on “usage_count” descending. The more an ingredient is used, the higher on the list it will appear. Updating this usage_count field is done with a PUT endpoint which will be called after each of the above PUT endpoints.

Update Ingredient Document Incrementing “usage_count” Endpoint: PUT

/api/:collection/:ingredient/increment

Database Document Before:

```
{
  "_id": 65,
  "ingredient": "Matcha powder",
  "types": Array (2),
  "rank": 65,
  "usage_count": 86
}
```

Example Request:

The screenshot shows a REST client interface with the following details:

- URL:** `http://localhost:3000/api/dessert_ingredients/65/increment`
- Method:** PUT
- Body:** 1
- Status:** 200 OK (35 ms)
- Response Body (JSON):**

```
{
  "message": "Usage count incremented for '65'.",
  "modifiedCount": 1
}
```

Database Document After:

```
{
  "_id": 65,
  "ingredient": "Matcha powder",
  "types": Array (2),
  "rank": 65,
  "usage_count": 87
}
```

Example Error (404 Not Found):

The screenshot shows a REST client interface with the following details:

- URL:** `http://localhost:3000/api/dessert_ingredients/1000/increment`
- Method:** PUT
- Body:** 1
- Status:** 404 Not Found (9 ms)
- Response Body (JSON):**

```
{
  "error": "Ingredient '1000' not found in dessert_ingredients."
}
```

Should a user decide to remove an ingredient from a recipe, they can click on that line in the recipe display and click the “Remove” button. When doing so, a DELETE endpoint removes the ingredient from the recipe.


Delete Ingredient Endpoint: DELETE /api/recipes/:id/ingredients



Database Before:



```
_id: 6
user_id: 1
name: "Beef Stroganoff"
category: "Main Courses"
type: "Beef"
▼ ingredients: Object
  ► Beef sirloin: Object
  ► Salt: Object
  ► Black pepper: Object
```

Example Request:




 http://localhost:3000/api/recipes/6/ingredients





DELETE  http://localhost:3000/api/recipes/6/ingredients

Params Auth Headers (9) **Body**  Scripts  Settings

raw  **JSON** 

```
1 {
2   |   "name": "Salt"
3 }
```

Body   **200 OK** 



 **JSON**   Preview  Visualization


```
1 {
2   |   "message": "Ingredient 'Salt' removed successfully",
3   |   "modifiedCount": 1
4 }
```



Database After:



```
_id: 6
user_id: 1
name: "Beef Stroganoff"
category: "Main Courses"
type: "Beef"
▼ ingredients: Object
  ► Beef sirloin: Object
  ► Black pepper: Object
```

Example Request (400 Bad Request):




 http://localhost:3000/api/recipes/6/ingredients 





DELETE  http://localhost:3000/api/recipes/6/ingredients

Params Auth Headers (9) **Body**  Scripts  Settings

raw  **JSON** 

```
1 {
2   |   "name": "Foo"
3 }
```

Body   **400 Bad Request**  6 m

 **JSON**   Preview  Visualization

```
1 {
2   |   "error": "Ingredient 'Foo' not found in the recipe."
3 }
```

At any time, the user can add instruction steps to the recipe by typing them in the provided text entry box. When they click the “Submit Step” button, a PUT endpoint adds the instruction to the recipe.

Add or Update Instructions to Recipe Document Endpoint: PUT /api/recipes/:id/instructions

Example Request:

The screenshot shows a REST client interface with the URL `http://localhost:3000/api/recipes/3/instructions`. The method is set to **PUT**. The request body is in **JSON** format and contains the following structure:

```
1 {
2   "push": {
3     "instructions": "Serve with Caramel or Chocolate Sauce"
4   }
5 }
```

The response status is **200 OK** with a response time of 9 ms. The response body is in **JSON** format and contains the following structure:

```
1 {
2   "message": "Instruction added successfully",
3   "modifiedCount": 1
4 }
```

Database Updated:

```
_id: 3
user_id: 1
name: "Vanilla Ice Cream"
category: "Desserts"
type: "Ice Cream"
▶ ingredients: Object
▼ instructions: Array (5)
  0: "Whisk together heavy cream, milk, sugar, vanilla ex
  1: "Chill mixture in the refrigerator for at least 2 hc
  2: "Pour into an ice cream maker and churn according to
  3: "Transfer to a container and freeze until firm."
  4: "Serve with Caramel or Chocolate Sauce"
```

Example Error (404 Not Found):

The screenshot shows a REST client interface with the URL `http://localhost:3000/api/recipes/12/instructions`. The method is set to **PUT**. The request body is in **JSON** format and contains the following structure:

```
1 {
2   "push": {
3     "instructions": "Serve with Caramel or Chocolate Sauce"
4   }
5 }
```

The response status is **404 Not Found** with a response time of 6 ms. The response body is in **JSON** format and contains the following structure:

```
1 {
2   "error": "Recipe not found."
3 }
```

Users will also be able to delete lines of instructions in much the same way. This utilizes a DELETE endpoint when the user clicks the “Remove” button in the instruction list.

Delete Recipe Instruction Endpoint: DELETE /api/recipes/:id/instructions

Database Before:

```
_id: 3
user_id: 1
name: "Vanilla Ice Cream"
category: "Desserts"
type: "Ice Cream"
ingredients: Object
instructions: Array (5)
  0: "Whisk together heavy cream, milk, sugar, vanilla"
  1: "Chill mixture in the refrigerator for at least 2"
  2: "Pour into an ice cream maker and churn according"
  3: "Transfer to a container and freeze until firm."
  4: "Serve with Caramel or Chocolate Sauce"
```

Example Request:

HTTP <http://localhost:3000/api/recipes/3/instructions>

DELETE <http://localhost:3000/api/recipes/3/instructions>

Params Auth Headers (9) Body ● Scripts ● Settings

raw JSON

```
1 {
2   |   "instructions": "Serve with Caramel or Chocolate Sauce"
3 }
```

Body 200 OK • 12 ms

{ } JSON Preview Visualization

```
1 {
2   |   "message": "Instruction removed successfully",
3   |   "modifiedCount": 1
4 }
```

Database After:

```
_id: 3
user_id: 1
name: "Vanilla Ice Cream"
category: "Desserts"
type: "Ice Cream"
ingredients: Object
instructions: Array (4)
  0: "Whisk together heavy cream, milk, sugar, vanilla extract,"
  1: "Chill mixture in the refrigerator for at least 2 hours."
  2: "Pour into an ice cream maker and churn according to manufa"
  3: "Transfer to a container and freeze until firm."
created at: 2025-02-18T12:45:00.000+00:00
```

Example Error (400 Bad Request):

HTTP <http://localhost:3000/api/recipes/3/instructions>

DELETE <http://localhost:3000/api/recipes/3/instructions>

Params Auth Headers (9) Body ● Scripts ● Settings

raw JSON

```
1 {
2   |   "instructions": "Serve with Caramel or Chocolate Sauce"
3 }
```

Body 400 Bad Request • 8 ms

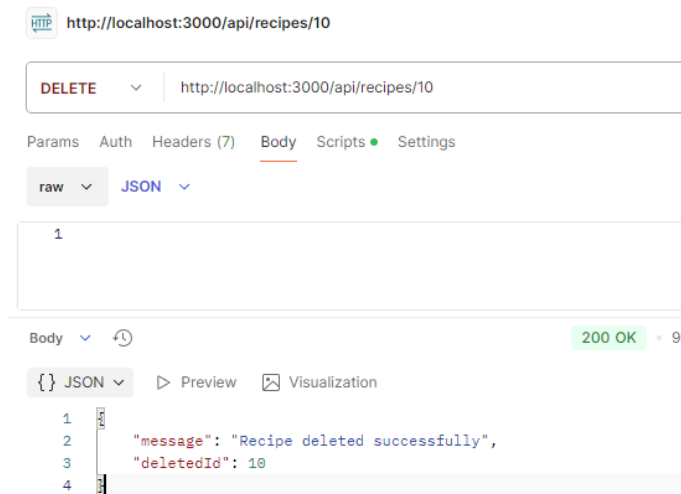
{ } JSON Preview Visualization

```
1 {
2   |   "error": "Instruction not found in recipe."
3 }
```

When displayed, a “Delete” recipe button will be provided to the user. It will only function on recipes where the `userId` field matches that of the user.

Delete Recipe Endpoint: DELETE /api/recipes/:id

Example Request:

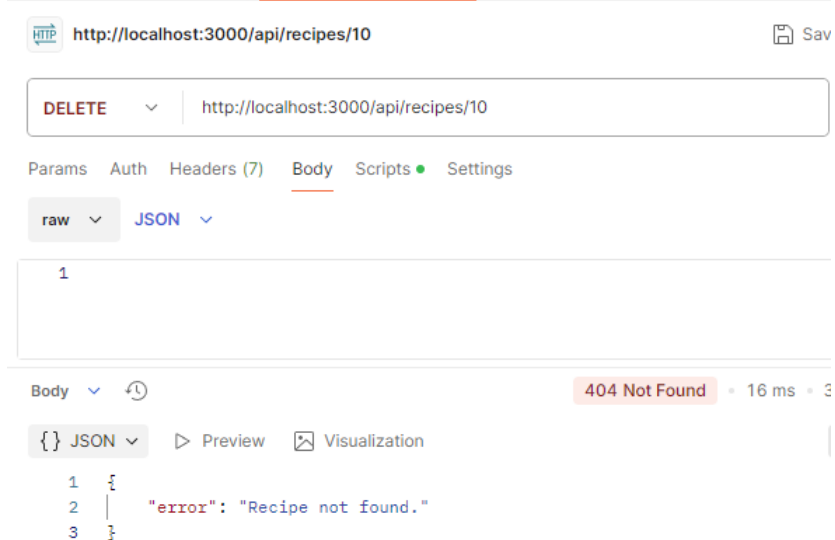


The screenshot shows a REST client interface with the following details:

- Method: DELETE
- URL: http://localhost:3000/api/recipes/10
- Body: 1
- Status: 200 OK
- Response Body (JSON):

```
{  "message": "Recipe deleted successfully",  "deletedId": 10}
```

Example Error (404 Not Found):



The screenshot shows a REST client interface with the following details:

- Method: DELETE
- URL: http://localhost:3000/api/recipes/10
- Body: 1
- Status: 404 Not Found
- Response Body (JSON):

```
{  "error": "Recipe not found."}
```

6. Conclusion

This service layer design provides a clear separation between the frontend UI and MongoDB database, ensuring:

- **Consistency:** Defined endpoints for data access.
- **Scalability:** Easy future modifications.
- **Maintainability:** Clean API structure for interacting with recipes, users, and ingredients.

The endpoints follow RESTful principles, ensuring predictable behaviors for retrieving, creating, updating, and deleting data.
