RecipeDB Voice Chatbot

Backend Documentation

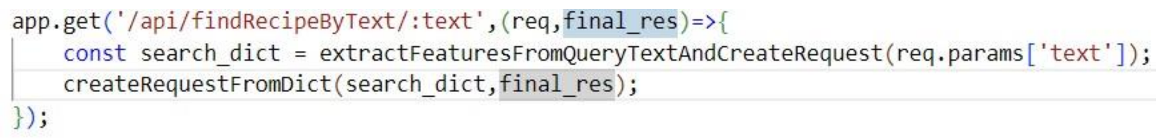
Parts of the documentation:

This is the documentation of the backend code of RecipeDB Voice Chatbot.

Let's say a user clicks on **submit** button after speaking his query. Then, we call the backend of Voice Chatbot on the endpoint "/api/findRecipeByText/{text}" (where text is the query which user spoke).

Suppose the query is: “Show me some Indian recipes containing onion and tomato that can be made in microwave”.

This query will be received in backend at the endpoint and it will give a call to the function `extractFeaturesFromQueryTextAndCreateRequest(text)`



```
app.get('/api/findRecipeByText/:text',(req,final_res)=>{
    const search_dict = extractFeaturesFromQueryTextAndCreateRequest(req.params['text']);
    createRequestFromDict(search_dict,final_res);
});
```

Let’s see how this function works:

Section-1

<u>Function</u> `extractFeaturesFromQueryTextAndCreateRequest():`

We first create an empty dictionary with following keys:

- Category
- Ingredient
- Cooking process
- Utensil
- Country
- Continent

Then we split our query text by spaces resulting in an array of words. In our case we will get [“Show”,”me”,”some”,”Indian”,”recipes”,”containing”,”onion”,”and”,”tomato”,”that”,”can”,”be”,”made”,”in”,”micro wave”]

For each word we call a function `findWordInWhichDict()` which basically returns a list of two elements. The first element is the **type** of word, and the second element is the **word** itself. If the word doesn’t belong to any type it will return -1. The explanation on how `findWordInWhichDict()` works is explained in **Section-2**.

- If the word is of type “**Ingredient**” or “**Category**” we also check if its previous word was a negation (“no”, ”not”, ”without”). If yes, then we add [query word,-1] in the dictionary otherwise we simply add query word.
- If the word is of any other type, then we simply add query word to our dictionary.
- If the returned value by findWordInWhichDict() is -1 then we do not add that word to the dictionary.

Once we have iterated over all the words, we return the dictionary. In our case, the dictionary would look like.

{“Category”:,

“Ingredient”: [onion, tomato],

“Cooking process”:,

“Utensil”: microwave,

“Country”: Indian,

“Continent”:

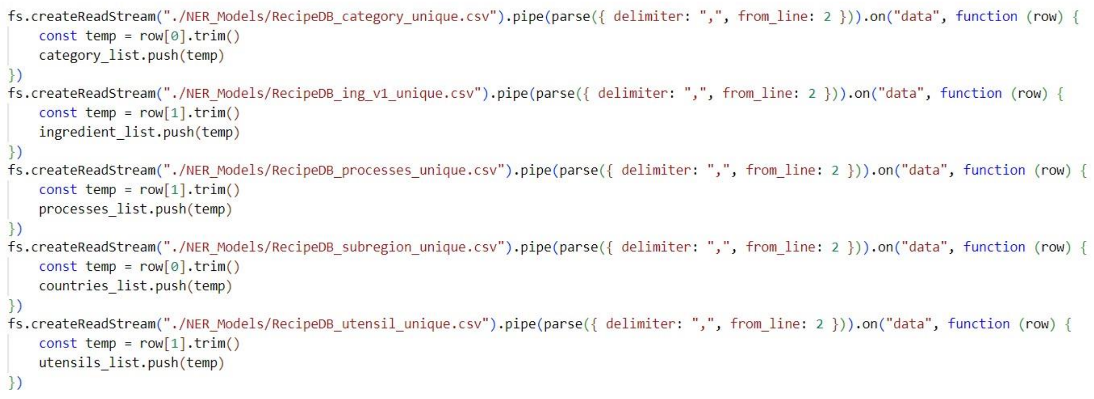
}

Section-2

<u>Function</u> `findWordInWhichDict():`

The aim of this function is to check whether a query word is present in RecipeDB or not. If it is present, we can give an API call to the recipeDB using that word to get a list of corresponding recipes.

We already have 5 lists containing words which are present in recipeDB.



```
fs.createReadStream("./NER_Models/RecipeDB_category_unique.csv").pipe(parse({ delimiter: ",", from_line: 2 })).on("data", function (row) {
    const temp = row[0].trim()
    category_list.push(temp)
})
fs.createReadStream("./NER_Models/RecipeDB_ing_v1_unique.csv").pipe(parse({ delimiter: ",", from_line: 2 })).on("data", function (row) {
    const temp = row[1].trim()
    ingredient_list.push(temp)
})
fs.createReadStream("./NER_Models/RecipeDB_processes_unique.csv").pipe(parse({ delimiter: ",", from_line: 2 })).on("data", function (row) {
    const temp = row[1].trim()
    processes_list.push(temp)
})
fs.createReadStream("./NER_Models/RecipeDB_subregion_unique.csv").pipe(parse({ delimiter: ",", from_line: 2 })).on("data", function (row) {
    const temp = row[0].trim()
    countries_list.push(temp)
})
fs.createReadStream("./NER_Models/RecipeDB_utensil_unique.csv").pipe(parse({ delimiter: ",", from_line: 2 })).on("data", function (row) {
    const temp = row[1].trim()
    utensils_list.push(temp)
})
```

These lists are created using the data available in CSV files in NER_Models folder.

Now, we have to check whether our query word is similar to any of the words in the 5 lists.

We start with first list and calculate levenshtein distance between all of the list words and the query word and append those distances in a new list. Then we find the minimum distance and store that in a variable min_val. The word corresponding to that min distance is most similar to our query word.

If the min_val is 0 then our query word exactly matches the list word. In this case we can return this word immediately.

It could also be possible that the user has made some error while typing or the speech to text converter has made some error so we can have some room for error by setting a threshold. The edit_length_to_be_waved variable takes care of that and can be changed according to personal requirements. It is currently set to 1. This means that if user has entered “Txmato” we can accept the query as “Tomato” as it differs by only 1. In this case also we return this word.

Similarly, we check in all 5 lists and return if any match is found. If not, we return -1.

Section-3

Now that we have a dictionary containing all the target words, we have to generate API request to RecipeDB.

Hence, we call `createRequestFromDict()` function.

Function `createRequestFromDict():`

Firstly, we have to delete those keys from the dictionary which does not have any value. In our case we delete Category, Cooking Process, Continent. Resulting dictionary looks like this:

{“Ingredient”: [onion, tomato],

“Utensil”: microwave,

“Country”: Indian,

}

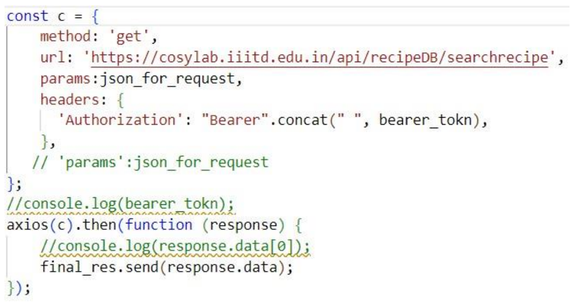
We would use following API from recipeDB to generate request:

`https://cosylab.iiitd.edu.in/api/recipeDB/searchrecipe`
It can take multiple parameters, however our interest of parameters are: country, ingredientUsed, ingredientNotUsed, cookingProcess, region, categoryUsed, categoryNotUsed.

We can pass these parameters in the form of a dictionary, where keys are the **parameter types** and values are the **words** which we have generated previously.

Now, we create a new dictionary json_for_request which will store all the parameters and values to be passed in the API request. We iterate over our previous dictionary search_dict and check if a key is “**Ingredient**” or “**Category**”. If yes, then we check if its value contains -1 or not. We need to check this because it tells us whether to use **Used** or **NotUsed** parameter. If -1 is present then NotUsed parameter will be used else otherwise. If the key is of some other type, then we can simply add that key-value pair in our json_for_request dictionary.

Now, we have to generate bearer token for our API request. Using axios, we generate bearer token and store that in a variable bearer_tokn. Once we get the token we can generate API request in the following format:



```
const c = {
    method: 'get',
    url: 'https://cosylab.iiitd.edu.in/api/recipeDB/searchrecipe',
    params:json_for_request,
    headers: {
      'Authorization': "Bearer".concat(" ", bearer_tokn),
    },
   // 'params':json_for_request
};
//console.log(bearer_tokn);
axios(c).then(function (response) {
    //console.log(response.data[0]);
    final_res.send(response.data);
});
```

The axios call generates the API request and we will get a list of recipes in the **response** variable. Using final_res.send() we send the generated list of recipes to frontend.

The frontend will take care of displaying the list of recipes in a proper format.

Section-4

The frontend will get a list of recipes and their corresponding IDs, using this ID the frontend will generate requests to the following endpoints:

`/api/recipeDB/recipeInfo/:recipe_id`
`/api/recipeDB/getingredientsbyrecipe/:recipe_id`
`/api/recipeDB/instructions/:recipe_id`

Now, we will see how do we handle requests at these endpoints.

Section -4.1

If the User clicks on recipe in the frontend, we have to display its details like: Name, image, ingredients, Instructions etc.

The basic details like name, image etc are retrieved using the following recipeDB API:

`https://cosylab.iiitd.edu.in/api/recipeDB/recipeInfo/`

In the backend, we give a function call to the RecipeInfo() function that handles this endpoint. In this function, we again generate a bearer token and use the RecipeID we got from frontend as parameter. Using axios() we get a response and send the response.data to the frontend.

Section -4.2

After the basic info, we need the list of ingredients. The frontend will send the recipe ID to the endpoint: `/api/recipeDB/getingredientsbyrecipe/:recipe_id`
In the backend, we will call the GetIngredientsByRecipe() function. Again, the bearer token is generated first and recipeID is set as the parameter of the RecipeDB API:
`https://cosylab.iiitd.edu.in/api/recipeDB/getingredientsbyrecipe/`

Using Axios, we get the response and we send this response to the frontend.

Section -4.3

Finally, we need the instructions. Using the same process as in previous two cases we can generate the instructions using the RecipeDB API: `https://cosylab.iiitd.edu.in/api/instructions/`

Before sending the data to frontend, we need to format it properly like setting the first letter of every instruction like to capital, ending each instruction with a full stop and inserting commas at relevant places. Once we have formatted the data, we send it to the frontend.