**CODE LAB II**

**Assessment 2**

|  |  |
| --- | --- |
| **Student Name** | Frem Fredric Baring Patalinghug |
| **Student ID** | 04-24-184 |
| **Tutor Name** | Ms. Amritha |
| **Repository Link** | [Eldeston/Advanced-Programming-Assessment-2](https://github.com/Eldeston/Advanced-Programming-Assessment-2) |
| **Video Link** |  |

**The Meal Database**

**Abstract**

This "MEALY DISPLAYINATOR 3000" is a cooking recipe viewer application using “TheMealDB” API. The application fetches user search queries from TheMealDB’s database to be used to serve and display the recipes to the user.

**Project Plan**

The project plan goes as follows in their respective order:

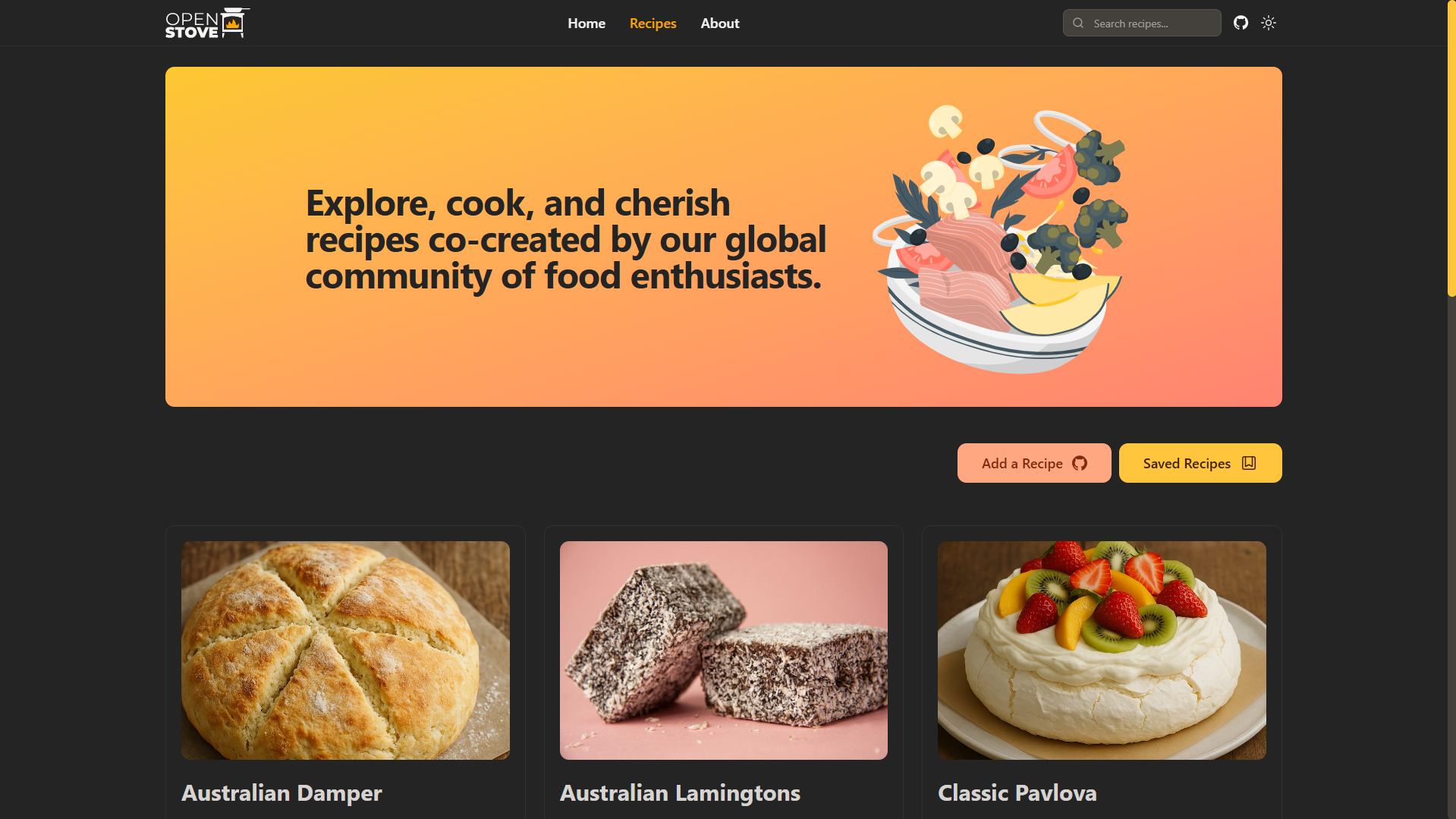
* Make an API class with necessary functions/methods to fetch data
* Make a base user interface in web application style:
  + A header widget to contain the application name & search widgets
  + A main widget to contain all the clickable recipe cards
  + A separate window to display the recipe’s ingredients & instructions
* Make a backend for the application’s features and connect it to the API

The application should also run with optimizations in mind by running the API and backend code in the background to improve user experience. This will be explained further in later sections.

**Evidence of Design**

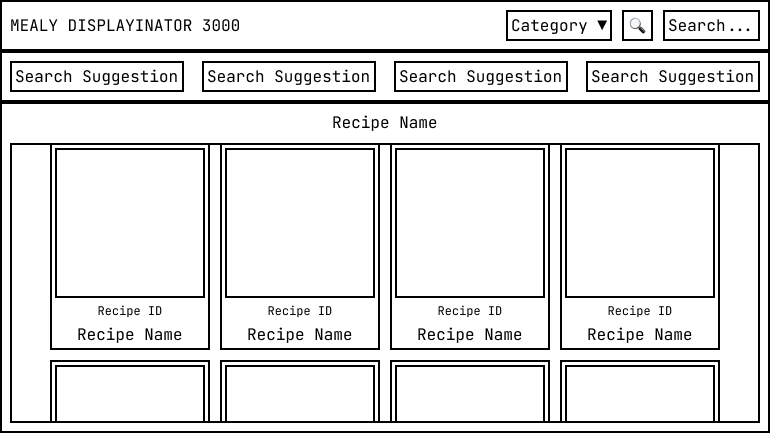
**Wireframe**

MEALY DISPLAYINATOR 3000's layout is inspired by OpenStove.org, an open source web project for storing cooking recipes. The website can be found at: <https://openstove.org/recipes>



**Main App Window**

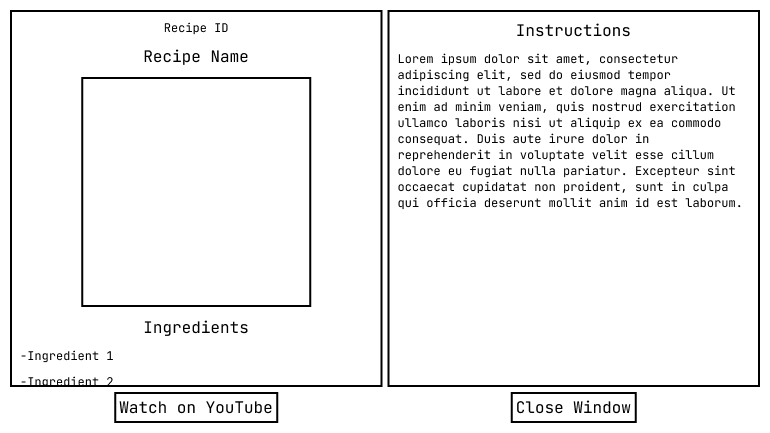
The main app window displays the core elements similar to a traditional website: a header and a main element. The header element contains the application name and search widgets. The search widgets include a search entry, button, and type. The header will also contain a suggestion frame to contain the top 4 search results. The main element contains a label and a container containing all the recipe cards with each card storing an image of the meal, ID, and name.



Wireframe of main application window.

**Top Level Window**

The top level window, or the popup window will open when a recipe card is clicked. The top level window will contain 2 main sections: the prerequisites window and an instructions window. The prerequisites window will contain the meal's name, image, and the required ingredients. The instructions window will contain the instructions on how to prepare and cook the meal. Additionally the top level window will contain buttons to view the recipe's guide on YouTube and a close window button. The necessary contents will be provided by the API.



Wireframe of top level (pop up) window.

**UML Diagram**

Here is a Unified Modeling Language (UML) Diagram of the application. Each card represents a class in the application and its relationships with other classes.

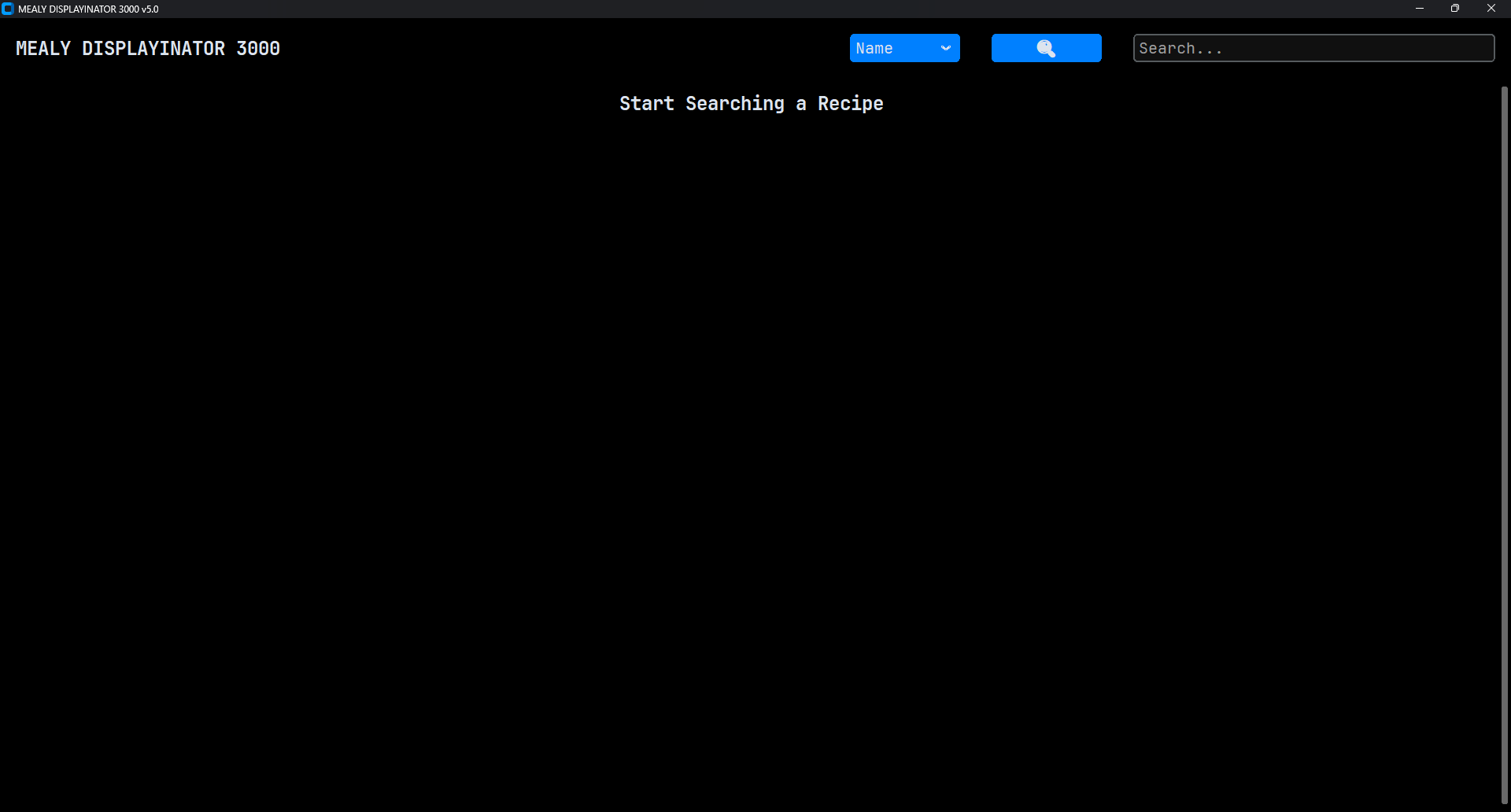


UML diagram of MEALY DISPLAYINATOR using Mermaid in Markdown.

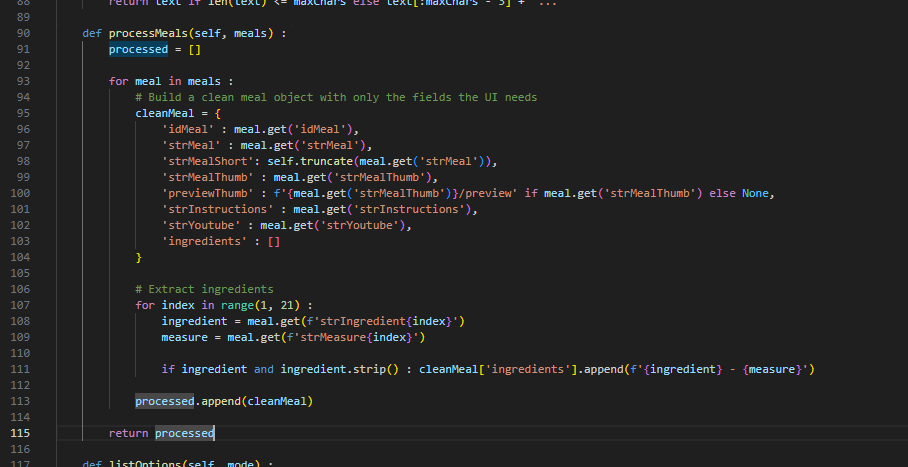
**Technical Description & Walkthrough**

MEALY DISPLAYINATOR 3000 uses Python 3.14.0 and the application uses the following packages:

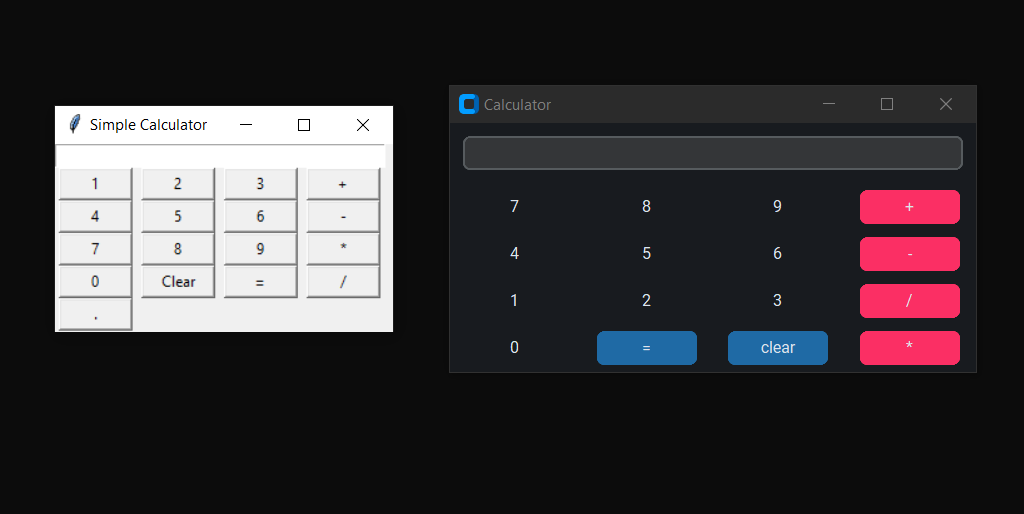
* Requests : For requesting API data from TheMealDB
* Threading : For fetching API data in the background
* Webbrowser : For opening URL links
* PIL (Pillow) : For image widgets
* IO (I/O) : For converting data
* CustomTKinter : For UI display



The API this application uses has several dedicated methods that makes data management easier. This includes API requests, search prompts, and data processing for "cleaning" data. An API request starts with the API's URL and the prompt syntax according to the API's documentation. The search prompt simply translates the user's input into the API's URL prompt syntax. Data processing for "cleaning" or processing requested API data into a simpler format that the application can use.

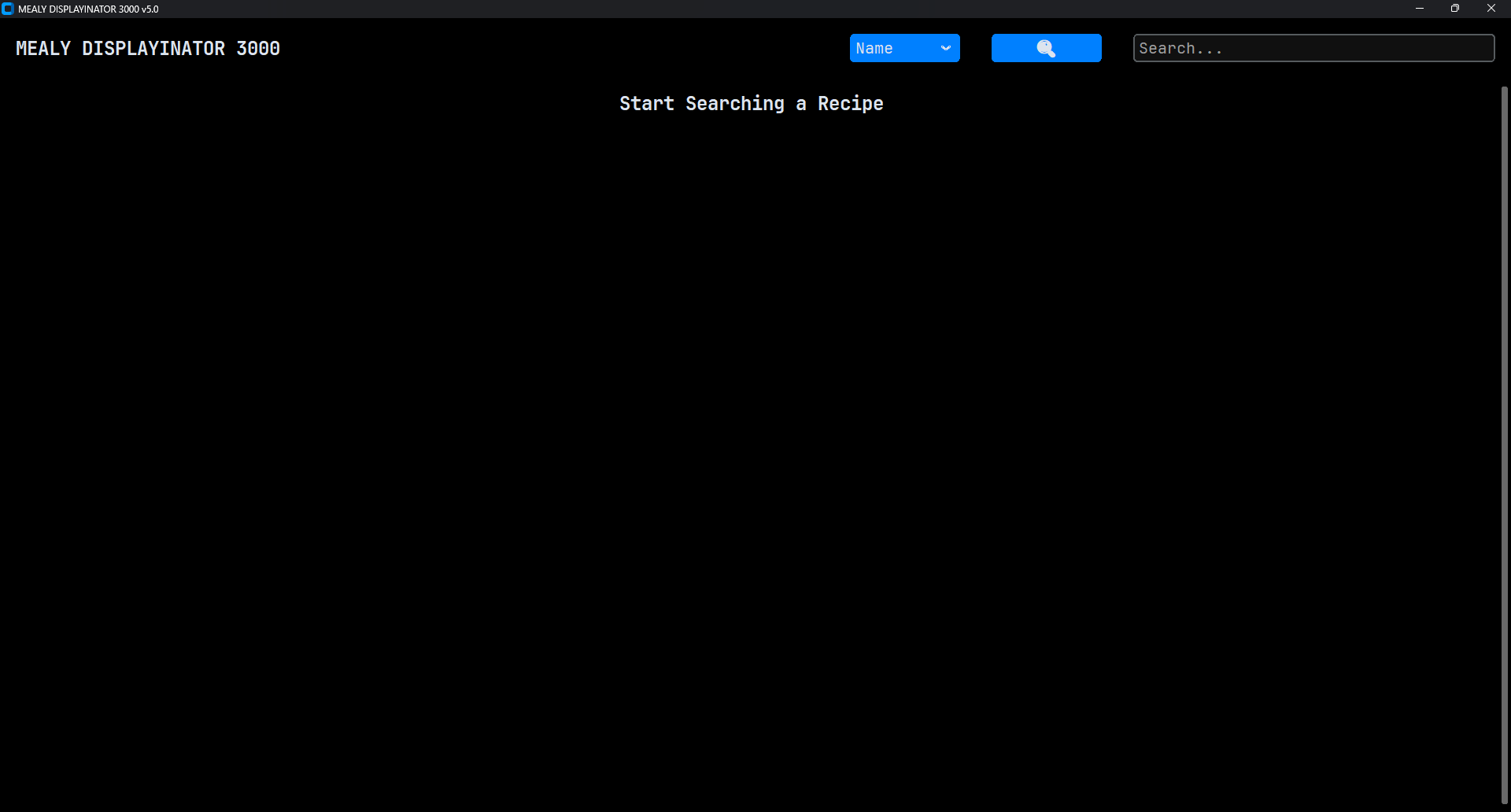
  
Process meals method

The application takes advantage of the CustomTKinter package instead of TKinter. The reason for this design decision is because CustomTKinter is more flexible in changing styles and has more widget options than TKinter. The widget classes HeaderUI, MainUI, CardUI, and RecipeUI all inherit CustomTKinter's widgets to adapt it to CustomTKinter's environment. CustomTKinter uses pillow image objects from PIL package to display images through CTkLabel widgets, but since we're fetching image data from a URL link, the package IO is used to convert raw image data that PIL can use to turn it into a pillow image object.

Python Hub (2023) CustomTKinter vs Tkinter Comparison Image [online image].

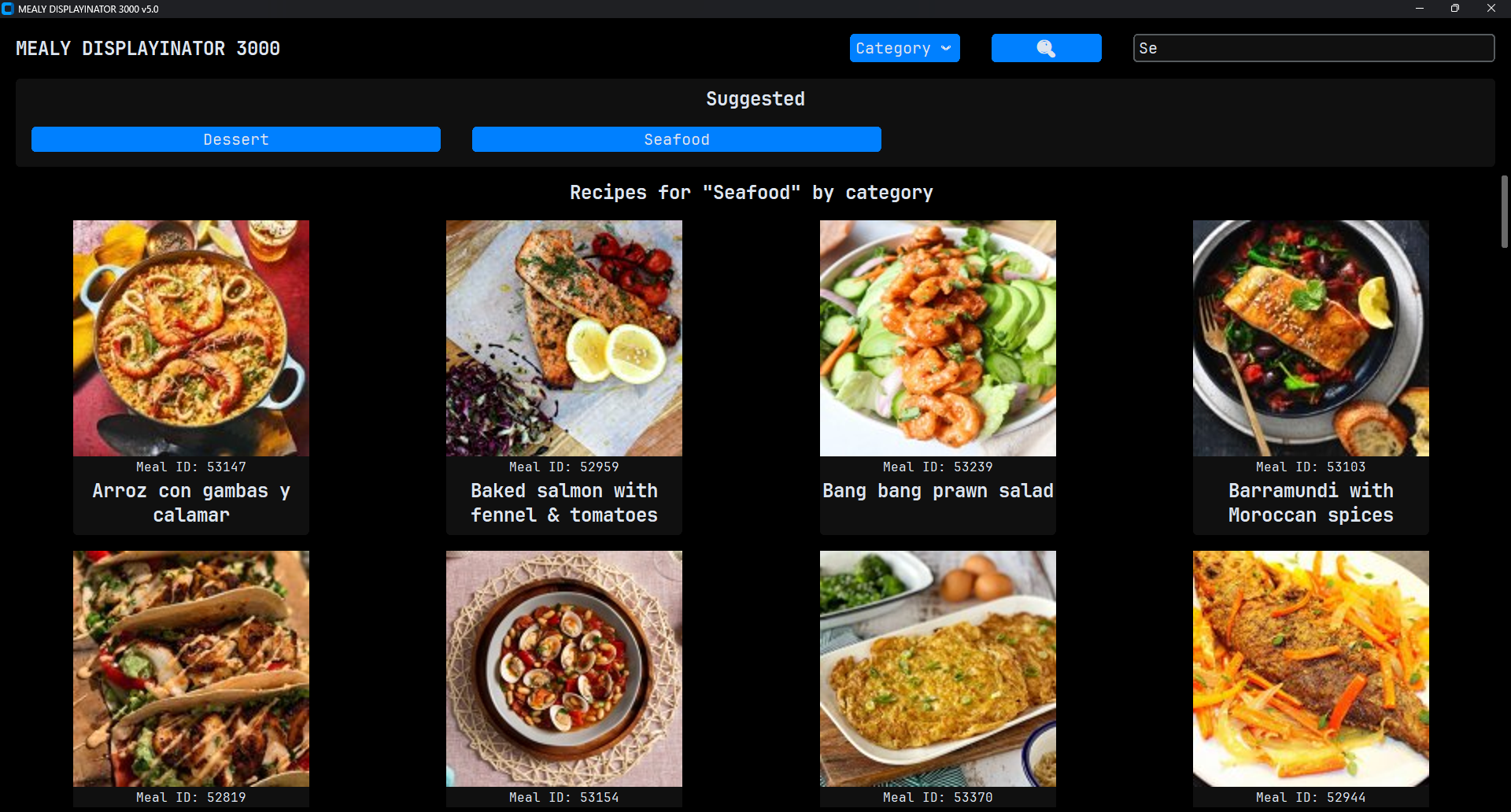
Available at: <https://python-hub.com/customtkinter-python-basics/> (Accessed: 31 December 2025).

As the main application class runs, it takes advantage of the Threading package to run the API requests in the background to load CustomTKinter's UI immediately upon application run. The user is met with a label asking the user to use the search bar to start searching for a meal by name, category, ingredient, or ID inside the option menu widget. As the user makes their prompt the suggestions frame starts populating with top 4 suggested search button options for the user. The user can either click on any of the suggested search options.



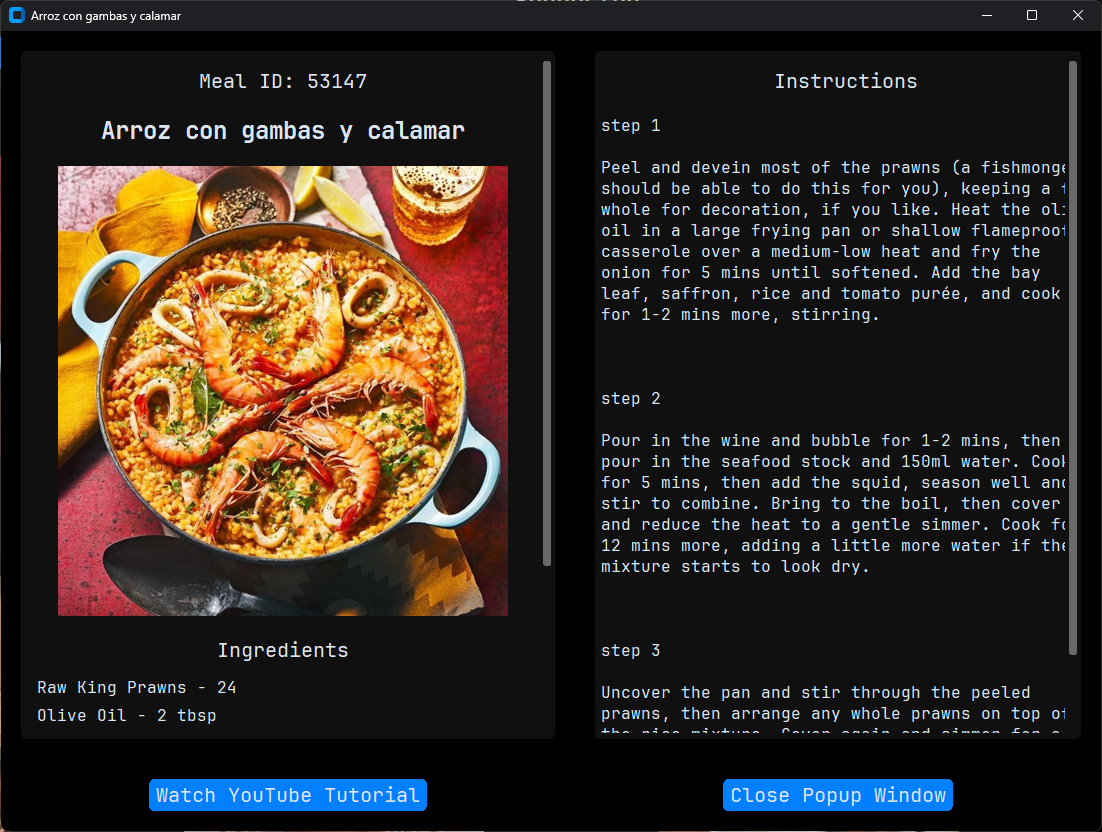
MEALY DISPLAYINATOR 3000’s Main Window UI.

When the user runs their search prompt by pressing enter in the search bar, clicking on the search button, or clicking on any of the available suggested options, the MainUI widgets starts loading as the application fetches data from the API based on the user's search prompt. Once the application receives the data, the MainUI frame starts populating with CardUI widgets each containing the meal's image, ID, and name.



Searching for “Seafood” in MEALY DISPLAYINATOR 3000.

Each card stores the meal's full data. Clicking on a card passes the data to the top level window and opens the window as a popup to display the full meal data to the user. This includes the card's initial data, ingredients, instructions, and YouTube URL button which opens the user's browser using the Webbrowser package.



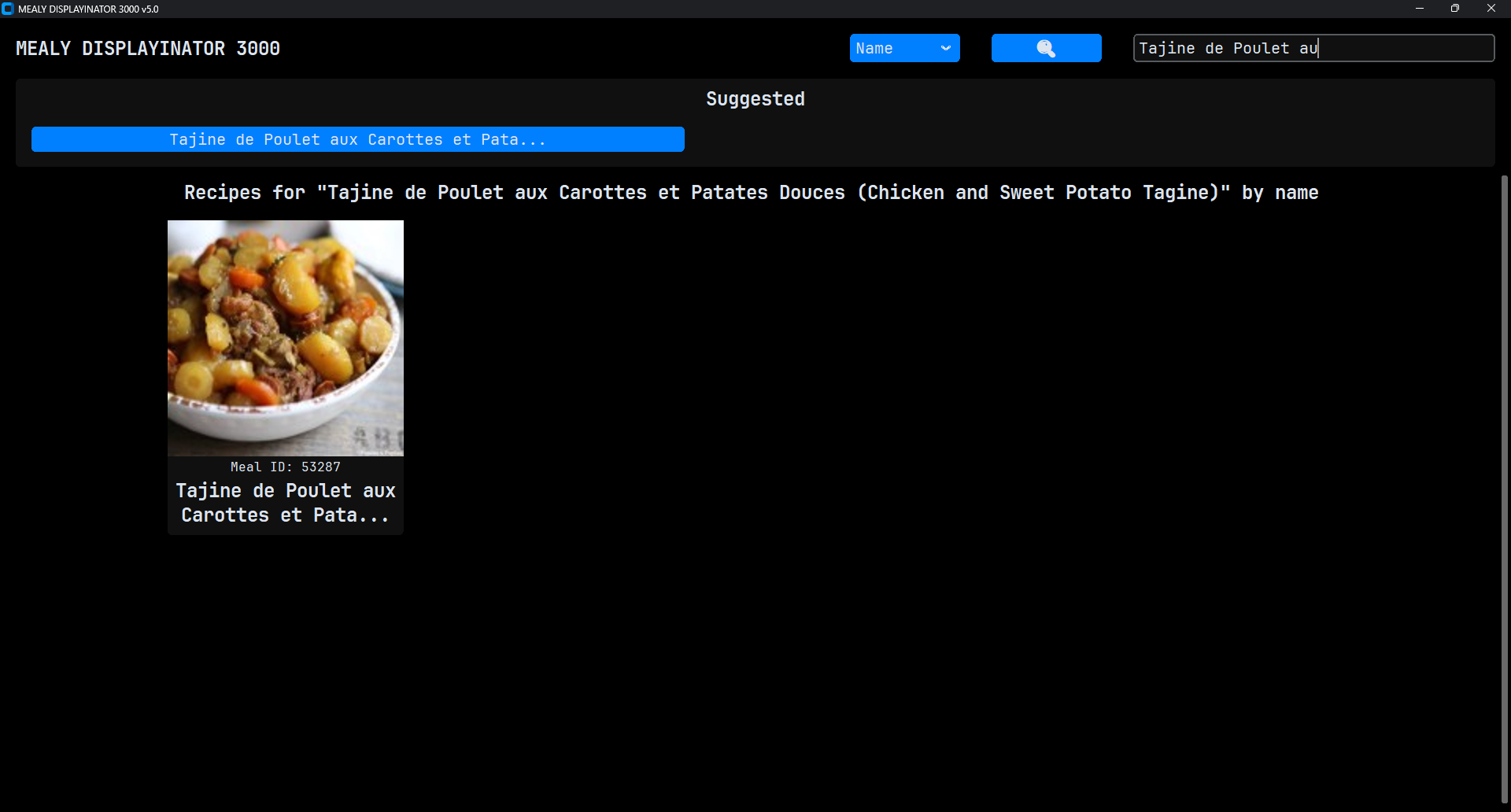
Opening a recipe card in MEALY DISPLAYINATOR 3000.

**Testing**

The application has undergone several revisions up to version 5.0. The initial versions began with the initial layout of the widgets testing the responsive layout. The API is later added and used to connect to the widget's functions. The application is tested against its responsive UI, error handling, optimizations.

**Responsive UI**

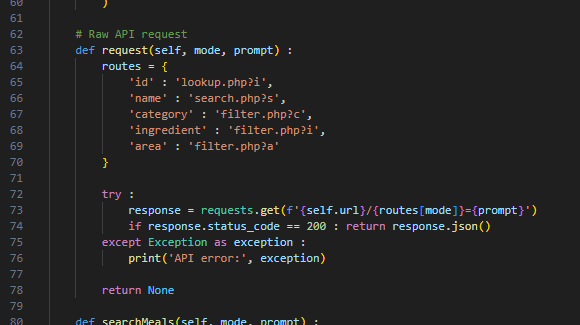
Labels that are too wide to fit in a widget are wrapped by a maximum width limit in pixels. This is applied to the CardUI widget and RecipeUI window. However, some of the meal names are still too large to be contained in a widget, so there is an additional value added into the cleaned meal that is the truncated version of the name. This would turn "Tajine de Poulet aux Carottes et Patates Douces (Chicken and Sweet Potato Tagine)" limited to 42 characters and ends with three dots: "Tajine de Poulet aux Carottes et Pata..."



MEALY DISPLAYINATOR 3000 handling word wrapping and truncation.

**Error Handling**

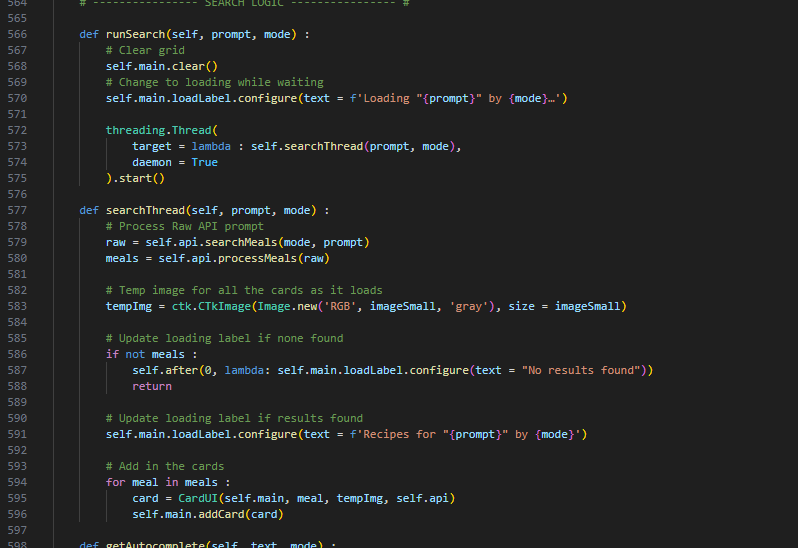
When it comes to an API error in a request, the request will simply return an empty response and throw an error in the console. If no data is received, the widget remains unaffected and the application runs normally and does not crash prematurely.



Raw request method, all API requests depend on it

**Optimization**

API requests are optimized to run in the background to load the UI first instead of waiting to receive requests using threading. All API requests will run asynchronously and widget updates will run synchronously. Loading indicators are placed to let the user now the application is currently fetching API data.



Threading optimization applied in search method

**Critical Reflection**

MEALY DISPLAYINATOR 3000's development improved and solidified my understanding of Object Oriented Programming's (OOP) benifits. Making a API class helped me manage and organize my functions/methods and improved my abstract thinking. I learned how to create my own widget by inheriting CustomTKinter's available widgets to adapt to CustomTKinter's environment. This makes my workflow similar to how I write in HTML and manage design decisions easily.

Writing with OOP also helped me make shorter and more modular code on top of what I have already learned with functions/methods. Even though this assessment is focused on Python, I can see where OOP can be applied in other languages such as the JavaScript language and any variation of the C language. Writing with OOP also helped me diagnose errors as well as documenting the code. The console can tell which class and method has the issue rather than scrolling through lines of code. Each class and its methods can be visualized clearly as I document my code.

**Appendix**

# -------------------------------- HIERARCHY -------------------------------- #

# MealAPI - API

# CardUI - Widget

# HeaderUI - Widget

# MainUI - Widget

# RecipeUI - Top Level

# Application - Main Window

# -------------------------------- IMPORTS -------------------------------- #

# Imports requests for getting API data

import requests

# Imports threading to load application early while downloading data

import threading

# To open browser links

import webbrowser

# Imports pillow for images

from PIL import Image

# Imports BytesIO for conversion

from io import BytesIO

# Install customtkinter

# python -m pip install customtkinter

import customtkinter as ctk

# -------------------------------- STYLING -------------------------------- #

applicationName = 'MEALY DISPLAYINATOR 3000'

applicationVersion = 5.0

smallPadding = 10

bigPadding = 20

smallFont = ('JetBrains Mono', 16)

mediumFont = ('JetBrains Mono', 20)

bigFont = ('JetBrains Mono', 24, 'bold')

imageSmall = (300, 300)

imageBig = (450, 450)

accentCol = "#0080FF"

backGroundCol = "#000000"

foreGroundCol = "#101010"

# -------------------------------- CORE API -------------------------------- #

class MealAPI :

    def \_\_init\_\_(self) :

        self.url = 'https://www.themealdb.com/api/json/v1/1'

        print(

f'''

The Meal DB : Free Recipe API

The API and site will always remain free at point of access.

URL: {self.url}

'''

        )

    # Raw API request

    def request(self, mode, prompt) :

        routes = {

            'id' : 'lookup.php?i',

            'name' : 'search.php?s',

            'category' : 'filter.php?c',

            'ingredient' : 'filter.php?i',

            'area' : 'filter.php?a'

        }

        try :

            response = requests.get(f'{self.url}/{routes[mode]}={prompt}')

            if response.status\_code == 200 : return response.json()

        except Exception as exception :

            print('API error:', exception)

        return None

    def searchMeals(self, mode, prompt) :

        data = self.request(mode, prompt)

        if not (data and data.get('meals')) : return []

        return data['meals']

    def truncate(self, text, maxChars = 40):

        if not text : return ""

        return text if len(text) <= maxChars else text[:maxChars - 3] + "..."

    def processMeals(self, meals) :

        processed = []

        for meal in meals :

            # Build a clean meal object with only the fields the UI needs

            cleanMeal = {

                'idMeal' : meal.get('idMeal'),

                'strMeal' : meal.get('strMeal'),

                'strMealShort': self.truncate(meal.get('strMeal')),

                'strMealThumb' : meal.get('strMealThumb'),

                'previewThumb' : f'{meal.get('strMealThumb')}/preview' if meal.get('strMealThumb') else None,

                'strInstructions' : meal.get('strInstructions'),

                'strYoutube' : meal.get('strYoutube'),

                'ingredients' : []

            }

            # Extract ingredients

            for index in range(1, 21) :

                ingredient = meal.get(f'strIngredient{index}')

                measure = meal.get(f'strMeasure{index}')

                if ingredient and ingredient.strip() : cleanMeal['ingredients'].append(f'{ingredient} - {measure}')

            processed.append(cleanMeal)

        return processed

    def listOptions(self, mode) :

        routes = {

            'category': 'list.php?c=list',

            'ingredient': 'list.php?i=list',

            'area': 'list.php?a=list'

        }

        if mode not in routes:

            print(f'Invalid list mode: {mode}')

            return []

        try :

            response = requests.get(f'{self.url}/{routes[mode]}')

            if response.status\_code == 200 :

                data = response.json()

                # All three endpoints return a list under "meals"

                items = data.get('meals', [])

                # Normalize output to a simple list of strings

                if mode == 'category' : return [item['strCategory'] for item in items]

                if mode == 'ingredient' : return [item['strIngredient'] for item in items]

                if mode == 'area' : return [item['strArea'] for item in items]

        except Exception as exception :

            print('API error:', exception)

        return []

# -------------------------------- CARD UI -------------------------------- #

class CardUI(ctk.CTkFrame) :

    def \_\_init\_\_(self, parent, mealData, placeHolder, api : MealAPI) :

        super().\_\_init\_\_(parent, fg\_color = foreGroundCol, width = 300, height = 400, cursor = "hand2")

        self.pack\_propagate(False)

        self.api = api

        self.mealData = mealData

        # Image label

        self.imgLabel = ctk.CTkLabel(self, image = placeHolder, text = '', cursor = "hand2")

        self.imgLabel.pack()

        # ID label

        ctk.CTkLabel(

            self,

            text = f'Meal ID: {mealData['idMeal']}',

            font = smallFont,

            wraplength = 300

        ).pack()

        # Title

        ctk.CTkLabel(

            self,

            text = mealData['strMealShort'],

            font = bigFont,

            wraplength = 300

        ).pack()

        # Bind click to open recipe popup

        self.bind('<Button-1>', lambda event : self.openFullRecipe())

        self.imgLabel.bind('<Button-1>', lambda event : self.openFullRecipe())

        mealThumbUrl = mealData.get('previewThumb')

        if mealThumbUrl :

            threading.Thread(

                target = lambda : self.loadImageAsync(mealThumbUrl),

                daemon = True

            ).start()

    def openFullRecipe(self) :

        # Fetch full recipe details

        full = self.api.searchMeals('id', self.mealData['idMeal'])

        full = self.api.processMeals(full)

        if full : RecipeUI(full[0])

    # Replaces placeholder image with the actual image as it loads in the background

    def loadImageAsync(self, url) :

        try :

            response = requests.get(url)

            imgPIL = Image.open(BytesIO(response.content))

            imgTK = ctk.CTkImage(imgPIL, size = imageSmall)

            # Update image

            self.imgLabel.configure(image = imgTK)

        except Exception as exception :

            print('Failed to load image:', url, exception)

# -------------------------------- CARD GRID UI -------------------------------- #

class MainUI(ctk.CTkScrollableFrame) :

    def \_\_init\_\_(self, parent, maxColumns = 4, indexOffset = 0) :

        super().\_\_init\_\_(parent, fg\_color = "transparent")

        self.maxColumns = maxColumns

        self.indexOffset = indexOffset

        self.cards = []

        # Loading/Recipe label

        self.loadLabel = ctk.CTkLabel(

            self,

            text = 'Start Searching a Recipe',

            font = bigFont

        )

        self.loadLabel.grid(row = 0, column = 0, columnspan = maxColumns, padx = smallPadding, pady = smallPadding)

        # Configure responsive columns

        for col in range(maxColumns) :

            self.grid\_columnconfigure(col, weight = 1)

    # Clears all cards

    def clear(self) :

        for card in self.cards : card.destroy()

        self.cards.clear()

    # Adds a CardUI widget and places it in the grid

    def addCard(self, card) :

        index = len(self.cards) + self.indexOffset

        row = (index // self.maxColumns) + 1 # +1 because row 0 is the label

        col = index % self.maxColumns

        card.grid(row = row, column = col, padx = smallPadding, pady = smallPadding)

        self.cards.append(card)

# -------------------------------- RECIPE UI -------------------------------- #

class RecipeUI(ctk.CTkToplevel) :

    def \_\_init\_\_(self, meal) :

        super().\_\_init\_\_(fg\_color = backGroundCol)

        self.title(meal['strMeal'])

        self.geometry('1100x800')

        # Focuses all the input into this window

        self.grab\_set()

        # Configure layout (two-column layout)

        self.grid\_rowconfigure(0, weight = 1)

        self.grid\_columnconfigure(0, weight = 1)

        self.grid\_columnconfigure(1, weight = 1)

        # ---------------- MAIN FRAME (Left) ---------------- #

        mainFrame = ctk.CTkScrollableFrame(self, fg\_color = foreGroundCol)

        mainFrame.grid(row = 0, column = 0, sticky = 'nsew', padx = bigPadding, pady = bigPadding)

        mainFrame.grid\_columnconfigure(0, weight = 1)

        # Placeholder image

        self.tempImg = ctk.CTkImage(Image.new('RGB', imageBig, 'gray'), size = imageBig)

        # ID

        ctk.CTkLabel(

            mainFrame,

            text = f'Meal ID: {meal['idMeal']}',

            font = mediumFont,

            wraplength = 400

        ).grid(row = 0, column = 0, pady = smallPadding)

        # Meal Name

        ctk.CTkLabel(

            mainFrame,

            text = meal['strMeal'],

            font = bigFont,

            wraplength = 400

        ).grid(row = 1, column = 0, pady = smallPadding)

        # Image placeholder

        self.imgLabel = ctk.CTkLabel(mainFrame, image = self.tempImg, text = '')

        self.imgLabel.grid(row = 2, column = 0, pady = smallPadding)

        # Load image asynchronously

        if meal.get('strMealThumb') :

            threading.Thread(

                target = lambda : self.loadImageAsync(meal['strMealThumb']),

                daemon = True

            ).start()

        # Ingredients (Looped)

        ingredients = meal.get('ingredients', [])

        if ingredients :

            ctk.CTkLabel(

                mainFrame,

                text = 'Ingredients',

                font = mediumFont

            ).grid(row = 3, column = 0, pady = smallPadding)

            for index, item in enumerate(ingredients) :

                ctk.CTkLabel(

                    mainFrame,

                    text = item,

                    font = smallFont,

                    anchor = 'w',

                    justify = 'left'

                ).grid(row = 4 + index, column = 0, sticky = 'w', padx = smallPadding)

        # ---------------- INSTRUCTIONS FRAME (Right) ---------------- #

        instructionFrame = ctk.CTkScrollableFrame(self, fg\_color = foreGroundCol)

        instructionFrame.grid(row = 0, column = 1, sticky = 'nswe', padx = bigPadding, pady = bigPadding)

        ctk.CTkLabel(

            instructionFrame,

            text = 'Instructions',

            font = mediumFont

        ).grid(row = 0, column = 0, pady = smallPadding)

        self.instructions\_label = ctk.CTkLabel(

            instructionFrame,

            text = meal.get('strInstructions', 'No instructions available.'),

            font = smallFont,

            wraplength = 500,

            justify = 'left'

        )

        self.instructions\_label.grid(row = 1, column = 0, pady = smallPadding, sticky = 'nw')

        # YouTube Button, display if link is available

        if meal.get('strYoutube') :

            ctk.CTkButton(

                self,

                text = 'Watch YouTube Tutorial',

                font = mediumFont,

                fg\_color = accentCol,

                hover\_color = accentCol,

                command = lambda : webbrowser.open(meal['strYoutube'])

            ).grid(row = 1, column = 0, padx = bigPadding, pady = bigPadding)

        # Close Button

        ctk.CTkButton(

            self,

            text = 'Close Popup Window',

            font = mediumFont,

            fg\_color = accentCol,

            hover\_color = accentCol,

            command = self.destroy

        ).grid(row = 1, column = 1, padx = bigPadding, pady = bigPadding)

    # ---------------- Async loaders ---------------- #

    def loadImageAsync(self, url) :

        try :

            response = requests.get(url)

            imgPIL = Image.open(BytesIO(response.content))

            imgTK = ctk.CTkImage(imgPIL, size = imageBig)

            self.imgLabel.configure(image = imgTK, text = '')

        except Exception as exception :

            print('Failed to load recipe image :', exception)

# -------------------------------- HEADER UI -------------------------------- #

class HeaderUI(ctk.CTkFrame) :

    def \_\_init\_\_(self, parent, getSuggestions, runSearch) :

        super().\_\_init\_\_(parent, fg\_color = "transparent")

        self.fullSuggestions = []

        self.runSearch = runSearch

        self.getSuggestions = getSuggestions

        self.grid\_columnconfigure(0, weight = 1)

        # Track last typed text to avoid race conditions

        self.lastQuery = ""

        # ---------------- TITLE ---------------- #

        ctk.CTkLabel(

            self,

            text = applicationName,

            font = bigFont

        ).grid(row = 0, column = 0, padx = bigPadding, pady = bigPadding, sticky = 'nsw')

        # Mode menu

        self.modeMenu = ctk.CTkOptionMenu(

            self,

            values = ['Name', 'Category', 'Ingredient', 'Area', 'ID'],

            font = mediumFont,

            dropdown\_font = mediumFont,

            fg\_color = accentCol,

            button\_color = accentCol,

            button\_hover\_color = accentCol

        )

        self.modeMenu.grid(row = 0, column = 2, padx = bigPadding, pady = bigPadding, sticky = 'nse')

        self.modeMenu.configure(command = lambda e : self.refreshAutocomplete())

        # Search button

        self.searchButton = ctk.CTkButton(

            self,

            text = '🔍',

            font = bigFont,

            fg\_color = accentCol,

            hover\_color = accentCol

        )

        self.searchButton.grid(row = 0, column = 3, padx = bigPadding, pady = bigPadding, sticky = 'nse')

        # Search bar

        self.searchBar = ctk.CTkEntry(

            self,

            placeholder\_text = 'Search...',

            font = mediumFont,

            fg\_color = foreGroundCol

        )

        self.searchBar.grid(row = 0, column = 4, ipadx = bigPadding \* 8, padx = bigPadding, pady = bigPadding, sticky = 'nswe')

        # Bind typing event

        self.searchBar.bind('<KeyRelease>', lambda e : self.refreshAutocomplete())

        # ---------------- AUTOCOMPLETE FRAME ---------------- #

        self.autoFrame = ctk.CTkFrame(self, fg\_color = foreGroundCol)

        self.autoFrame.grid(row = 1, column = 0, columnspan = 5, sticky = 'nwe', padx = bigPadding)

        self.autoFrame.grid\_columnconfigure((0, 1, 2, 3), weight = 1)

        self.autoFrame.grid\_remove() # Hidden at startup

        # Suggested label

        self.suggestLabel = ctk.CTkLabel(

            self.autoFrame,

            text = 'Suggested',

            font = bigFont

        )

        self.suggestLabel.grid(row = 0, column = 0, columnspan = 4, pady = (smallPadding, 0))

        # Display 4 suggestion buttons, store them for later

        self.autoButtons = []

        for index in range(4) :

            btn = ctk.CTkButton(

                self.autoFrame,

                text = '',

                font = mediumFont,

                fg\_color = accentCol,

                hover\_color = accentCol,

                command = lambda i = index : self.selectSuggestion(i)

            )

            btn.grid(row = 1, column = index, padx = bigPadding, pady = bigPadding, sticky = 'we')

            self.autoButtons.append(btn)

    def modeGet(self) :

        return self.modeMenu.get().lower()

    def searchGet(self) :

        return self.searchBar.get().lower().strip()

    # ---------------- AUTOCOMPLETE LOGIC ---------------- #

    def refreshAutocomplete(self):

        mode = self.modeGet()

        text = self.searchGet()

        # Update last query

        self.lastQuery = text

        # Hide and exit early if the text is less than 2 characters

        if len(self.lastQuery) < 2 :

            self.autoFrame.grid\_remove()

            return

        threading.Thread(

            target = lambda: self.fetchSuggestions(text, mode),

            daemon = True

        ).start()

    def fetchSuggestions(self, text, mode) :

        # Ignore outdated threads

        if text != self.lastQuery : return

        # Ask the injected function for suggestions

        suggestions = self.getSuggestions(text, mode)

        # Push UI update to main thread

        self.after(0, lambda : self.showSuggestions(suggestions))

    def showSuggestions(self, suggestions) :

        self.fullSuggestions = suggestions

        # Ignore empty suggestions

        if not suggestions :

            self.autoFrame.grid\_remove()

            return

        self.autoFrame.grid()

        for index, btn in enumerate(self.autoButtons) :

            if index < len(suggestions) :

                options = suggestions[index]

                # CASE 2 : Category / Ingredient / Area → item is a string

                buttonText = options

                # CASE 1 : Name mode → item is a meal object

                if isinstance(options, dict) : buttonText = options['strMealShort']

                btn.configure(text = buttonText)

                btn.grid()

            else :

                btn.grid\_remove()

    def selectSuggestion(self, index) :

        options = self.fullSuggestions[index]

        # CASE 2 : Category / Ingredient / Area → item is a string

        buttonText = options

        # CASE 1 : Name mode → item is a meal object

        if isinstance(options, dict) : buttonText = options['strMeal']

        self.searchBar.delete(0, 'end')

        self.searchBar.insert(0, buttonText)

        self.autoFrame.grid\_remove()

        self.runSearch(buttonText, self.modeGet())

# -------------------------------- MAIN APPLICATION -------------------------------- #

class Application(ctk.CTk) :

    def \_\_init\_\_(self) :

        super().\_\_init\_\_(fg\_color = backGroundCol)

        self.geometry('360x360')

        self.after(0, self.wm\_state, 'zoomed')

        self.title(f'{applicationName} v{applicationVersion}')

        # Grid config to adapt to layout

        self.grid\_rowconfigure(0, weight = 0)

        self.grid\_rowconfigure(1, weight = 1)

        self.grid\_columnconfigure(0, weight = 1)

        self.api = MealAPI()

        # Header Widget

        header = HeaderUI(self, self.getAutocomplete, self.runSearch)

        header.grid(row = 0, column = 0, sticky = 'nwe')

        header.searchButton.configure(command = lambda : self.runSearch(header.searchGet(), header.modeGet()))

        header.searchBar.bind('<Return>', lambda event : self.runSearch(header.searchGet(), header.modeGet()))

        # Main Widget

        self.main = MainUI(self, 4, 0)

        self.main.grid(row = 1, column = 0, sticky = 'nsew')

    # ---------------- SEARCH LOGIC ---------------- #

    def runSearch(self, prompt, mode) :

        # Clear grid

        self.main.clear()

        # Change to loading while waiting

        self.main.loadLabel.configure(text = f'Loading "{prompt}" by {mode}…')

        threading.Thread(

            target = lambda : self.searchThread(prompt, mode),

            daemon = True

        ).start()

    def searchThread(self, prompt, mode) :

        # Process Raw API prompt

        raw = self.api.searchMeals(mode, prompt)

        meals = self.api.processMeals(raw)

        # Temp image for all the cards as it loads

        tempImg = ctk.CTkImage(Image.new('RGB', imageSmall, 'gray'), size = imageSmall)

        # Update loading label if none found

        if not meals :

            self.after(0, lambda: self.main.loadLabel.configure(text = "No results found"))

            return

        # Update loading label if results found

        self.main.loadLabel.configure(text = f'Recipes for "{prompt}" by {mode}')

        # Add in the cards

        for meal in meals :

            card = CardUI(self.main, meal, tempImg, self.api)

            self.main.addCard(card)

    def getAutocomplete(self, text, mode) :

        if mode in ['category', 'ingredient', 'area'] :

            # Use the API's list feature

            options = self.api.listOptions(mode)

            # Returns only the top 4 results

            return [item for item in options if text in item.lower()][:4]

        if mode == 'name' :

            # Simply use the API's existing search feature

            raw = self.api.searchMeals('name', text)

            meals = self.api.processMeals(raw)

            # Returns only the top 4 results

            return meals[:4]

        return []

# Run main application

Application().mainloop()