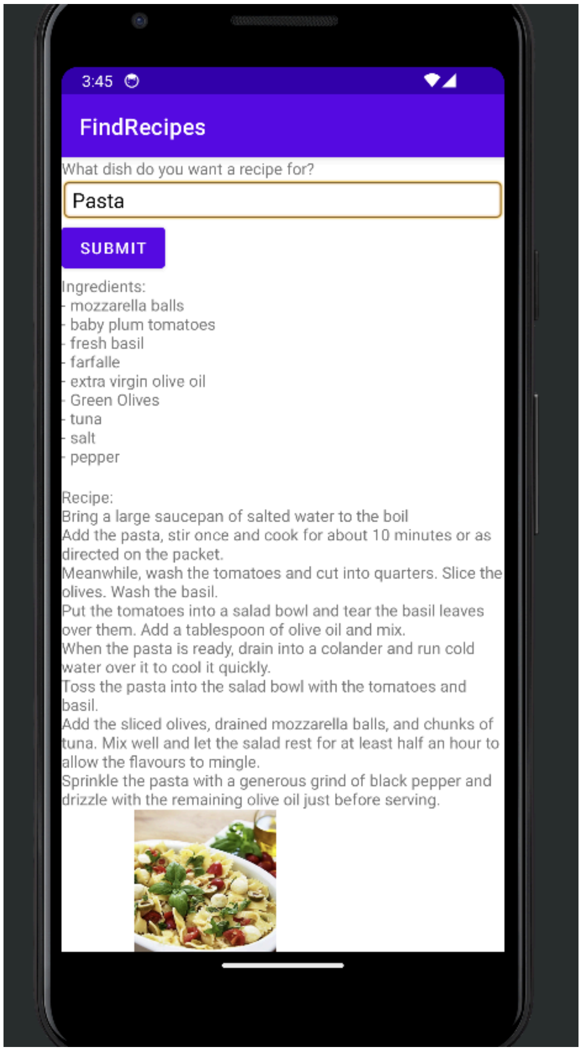
Priyanshi Singh
psingh3

Requirement 1

As we see we have implemented three views, 1. Image, 2. edit text and 3. text view. I have used https://www.themealdb.com/api.php api.



What dish do you want a recipe for?

Pasta

Ingredients:
- mozzarella balls
- baby plum tomatoes
- fresh basil
- farfalle
- extra virgin olive oil
- Green Olives
- tuna
- salt
- pepper

Recipe:
Bring a large saucepan of salted water to the boil
Add the pasta, stir once and cook for about 10 minutes or as directed on the packet.
Meanwhile, wash the tomatoes and cut into quarters. Slice the olives. Wash the basil.
Put the tomatoes into a salad bowl and tear the basil leaves over them. Add a tablespoon of olive oil and mix.
When the pasta is ready, drain into a colander and run cold water over it to cool it quickly.
Toss the pasta into the salad bowl with the tomatoes and basil.
Add the sliced olives, drained mozzarella balls, and chunks of tuna. Mix well and let the salad rest for at least half an hour to allow the flavours to mingle.
Sprinkle the pasta with a generous grind of black pepper and drizzle with the remaining olive oil just before serving.

The Android application developed meets all specified requirements as outlined:

1. Native Android Application:
- Implemented using Android SDK, which provides a native app experience.

2. Variety of Views:
- The application incorporates at least three different kinds of views:
  - `TextView` for displaying text such as ingredients and recipes.
  - `EditText` to allow users to input their search terms.

   - `ImageView` to display the picture fetched from the web service.

3. User Input:
  - Requires user input through an `EditText` view where users can enter search recipes for the dishes they are interested in.

4. HTTP Request:
  - Makes an HTTP GET request to a specified web service to fetch picture details based on the user's search term. This is handled in the `GetPicture` class using `HttpURLConnection`.

5. Parsing JSON Response:
  - Receives and parses a JSON formatted reply from the web service. The JSON object contains details such as picture URL, ingredients, and recipe which are then used to update the UI.

6. Displaying Information:
  - Displays new information to the user including the picture, ingredients, and recipe. This is dynamically shown in the app's UI depending on the response from the web service.

7. Repeatable Usage:
  - Users can repeatedly use the application to search for different pictures without needing to restart the app. Each search triggers a new HTTP request, and the UI is updated accordingly with new data each time.

InterestingPicture.java

```
/**
* @author Priyanshi Singh
* psingh3
*/
package com.example.dspicture;

// Importing necessary Android and Java classes.
import android.graphics.Bitmap;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.ImageView;
import android.widget.TextView;
import androidx.appcompat.app.AppCompatActivity;
import java.util.List;

/**
* An activity class that extends AppCompatActivity to provide UI and
functionality
```

```
* for searching and displaying picture details.
*/
public class InterestingPicture extends AppCompatActivity {

   /**
    * Called when the activity is starting. This is where most initialization
should go:
    * calling setContentView(int) to inflate the activity's UI, using
findViewById(int)
    * to programmatically interact with widgets, and setting up listeners.
    *
    * @param savedInstanceState If the activity is being re-initialized after
previously
    *                           being shut down then this Bundle contains the
most recent
    *                           data, otherwise it is null.
    */
   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       // Set the user interface layout for this Activity.
       // The layout file is defined in the project
res/layout/activity_main.xml file.
       setContentView(R.layout.activity_main);

       // Initialize the submit button from the layout.
       Button submitButton = findViewById(R.id.submit);
       // Set an OnClickListener to the button to handle user interactions.
       submitButton.setOnClickListener(view -> {
           // When the button is clicked, retrieve the text entered in the
EditText by the user.
           String searchTerm =
((EditText)findViewById(R.id.searchTerm)).getText().toString();
           // Create a new instance of GetPicture, passing the current instance
and the search term.
           GetPicture gp = new GetPicture(this, searchTerm);
           // Call the search method to execute the search operation.
           gp.search(this);
       });
   }

   /**
    * This method is a callback used by GetPicture to update the UI with the
picture and its details.
    * It updates the ImageView and TextViews with the picture, ingredients,
and recipe obtained.
    *
    * @param picture The bitmap image to display.
    * @param ingredients The list of ingredients associated with the picture.
```

```
     * @param recipe The text description of the recipe.
     */
    public void pictureAndDetailsReady(Bitmap picture, List<String> ingredients,
String recipe) {
        // Initialize UI components from the layout.
        ImageView pictureView = findViewById(R.id.interestingPicture);
        TextView ingredientsView = findViewById(R.id.ingredients);
        TextView recipeView = findViewById(R.id.recipe);

        // Check if a valid picture is received.
        if (picture != null) {
            // Set the picture to the ImageView and make it visible.
            pictureView.setImageBitmap(picture);
            pictureView.setVisibility(View.VISIBLE);

            // Build the ingredients text to display.
            StringBuilder ingredientsText = new StringBuilder("Ingredients:\n");
            for (String ingredient : ingredients) {
                ingredientsText.append("- ").append(ingredient).append("\n");
            }
            // Set the built string to the TextView for ingredients.
            ingredientsView.setText(ingredientsText.toString());

            // Set the recipe text to the TextView for the recipe.
            recipeView.setText("Recipe:\n" + recipe);
        } else {
            // If no picture is received, handle this case gracefully.
            pictureView.setVisibility(View.GONE); // Hide the ImageView.
            ingredientsView.setText("No picture available"); // Display a
fallback text.
            recipeView.setText("Recipe:\n" + recipe); // Still show the recipe
text.
        }
    }
}
```

Requirement 2

The implementation of the web service for the native Android application adheres to the specified requirements as follows:

1. Simple API Implementation:
   - A single-path API is implemented (`/submit`) within a servlet (`MealServlet`) that handles HTTP GET requests. This is configured using the `@WebServlet` annotation which clearly defines the URL pattern.

2. Reception of HTTP Requests:

- The servlet receives HTTP requests directly from the native Android application. It processes requests that include a meal name as a parameter, indicating the user's search term for meal information.

3. Execution of Business Logic:

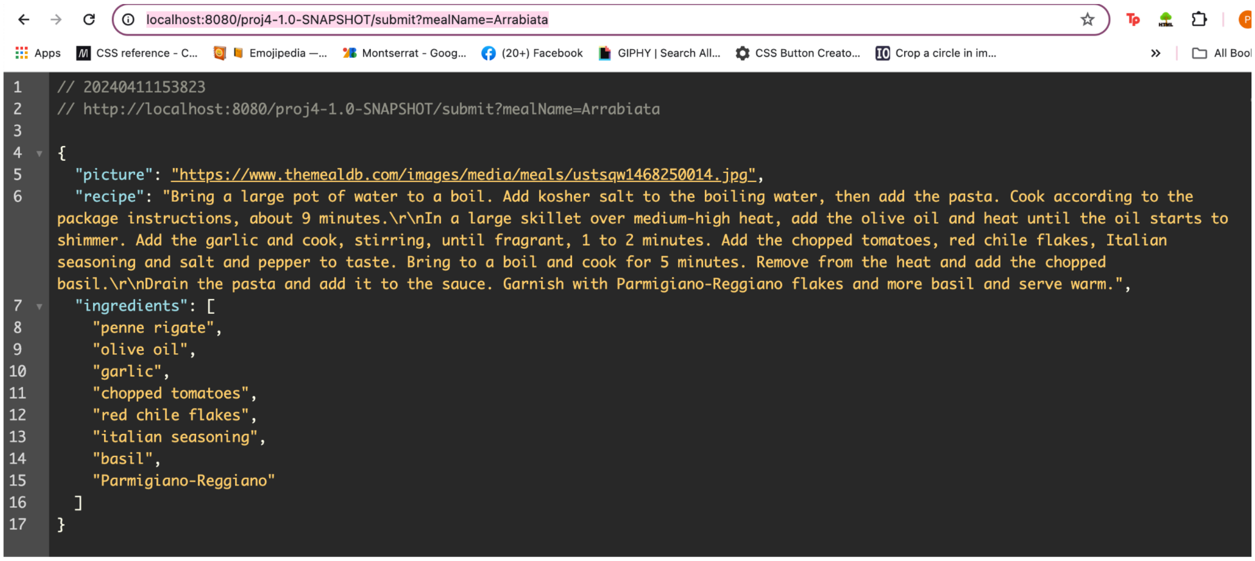
- The servlet performs several business logic operations:
  - Fetching Data: It makes an HTTP GET request to a third-party API ("TheMealDB") to retrieve meal data based on the search term provided.
  - Data Processing: Parses the JSON response from TheMealDB to extract and reformat necessary data (like picture URL, recipe, and ingredients).
  - Response Optimization: Constructs a simplified JSON object that only includes necessary data (picture, recipe, and ingredients) ensuring the mobile app does not need to perform excessive processing.

4. JSON Formatted Response:

- The servlet sends a JSON formatted response back to the Android application. This response is designed to include only essential data:
  - `picture`: URL of the meal's image.
  - `recipe`: Cooking instructions.
  - `ingredients`: A list of ingredients required for the meal.
- This approach ensures efficiency in data handling on the mobile device, adhering to the requirement of minimizing unnecessary data transfer and computation on the client side.

When a user inputs the name of a dish (eg. noodles, pasta) my web service uses the mealdb api to lookup the dish and store it's recipe, ingredients and picture in a json format and returns it. Eg. When user inputs mealName=Arrabiata on accessing the url:
http://localhost:8080/proj4-1.0-SNAPSHOT/submit?mealName=Arrabiata

We get



```
// 20240411153823
// http://localhost:8080/proj4-1.0-SNAPSHOT/submit?mealName=Arrabiata

{
  "picture": "https://www.themealdb.com/images/media/meals/ustsqw1468250014.jpg",
  "recipe": "Bring a large pot of water to a boil. Add kosher salt to the boiling water, then add the pasta. Cook according to the package instructions, about 9 minutes.\r\nIn a large skillet over medium-high heat, add the olive oil and heat until the oil starts to shimmer. Add the garlic and cook, stirring, until fragrant, 1 to 2 minutes. Add the chopped tomatoes, red chile flakes, Italian seasoning and salt and pepper to taste. Bring to a boil and cook for 5 minutes. Remove from the heat and add the chopped basil.\r\nDrain the pasta and add it to the sauce. Garnish with Parmigiano-Reggiano flakes and more basil and serve warm.",
  "ingredients": [
    "penne rigate",
    "olive oil",
    "garlic",
    "chopped tomatoes",
    "red chile flakes",
    "italian seasoning",
    "basil",
    "Parmigiano-Reggiano"
  ]
}
```

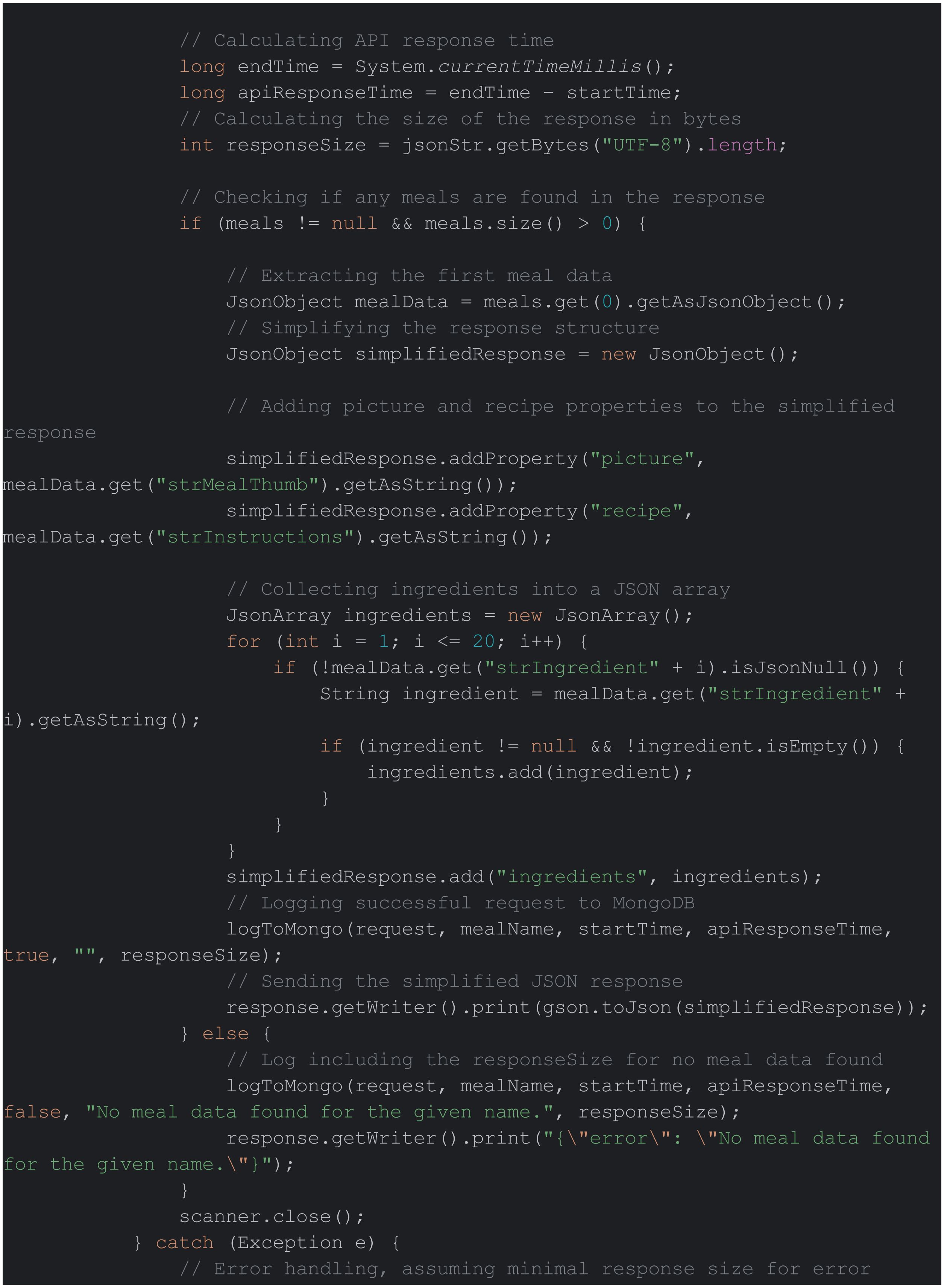
Along with this our MealServlet.java also logs several data points regarding our api request/responses in mongodb

5. Logging and Error Handling:
- Implements logging of each request to MongoDB, capturing details like the meal name, timestamp, API response time, user agent, and whether the request was successful or not.
- Efficient error handling and reporting: If the third-party API does not return a meal, or if any other error occurs (such as a connection error), the servlet responds with an appropriate error message. This ensures that the client application can gracefully handle failures.

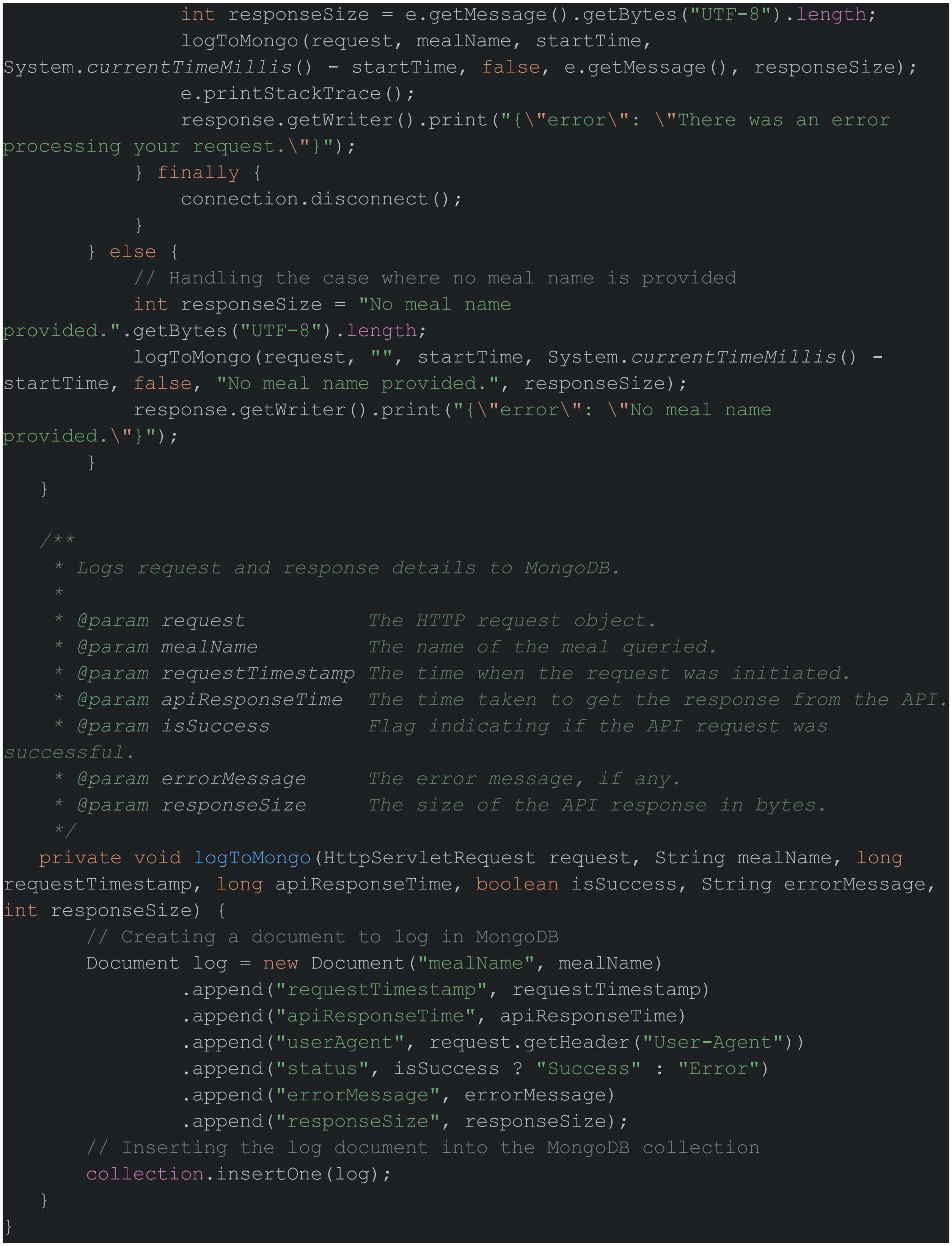
6. Usage of Servlets:
- The service uses Jakarta Servlet technology, specifically avoiding JAX-RS, to ensure compatibility and avoid deployment issues in environments like Docker containers, as specified.

MealServlet.java

```
/**
* @author: Priyanshi Singh
* psingh3
*/
package ds.proj4;

import com.google.gson.Gson;
import com.google.gson.JsonObject;
import com.google.gson.JsonArray;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.*;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import org.bson.Document;
import java.io.*;
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.Scanner;
/**
* This servlet handles meal information retrieval and logging of the request
details.
*/
@WebServlet(name = "MealServlet", urlPatterns = {"/submit"})
public class MealServlet extends HttpServlet {

   // MongoDB collection used to store logs
   private MongoCollection<Document> collection;

   public MealServlet() {
       // MongoDB connection string with credentials
       String connectionString =
"mongodb+srv://psingh3:Priyanshi23@cluster0.7yt1mug.mongodb.net/?retryWrites=tr
ue&w=majority&appName=Cluster0";
       // Establishing connection to the database
       MongoDatabase database =
MongoClients.create(connectionString).getDatabase("Project4");
       collection = database.getCollection("MealLogs");
       collection = database.getCollection("MealLogs");
   }
   /**
    * Handles GET requests by fetching meal details based on the provided meal
name.
    *
    * @param request  The HTTP request object.
    * @param response The HTTP response object to write the output to.
    */
   @Override
   protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws IOException {
       // Setting up the response format and encoding
       response.setContentType("application/json");
       response.setCharacterEncoding("UTF-8");

       // Tracking start time for performance logging
       long startTime = System.currentTimeMillis();
       // Getting the meal name parameter from the HTTP request
       String mealName = request.getParameter("mealName");

       // Check if the meal name parameter is provided and not empty
       if (mealName != null && !mealName.isEmpty()) {
           // API URL construction with the meal name query
           String apiUrl =
"https://www.themealdb.com/api/json/v1/1/search.php?s=" + mealName;
           // Opening HTTP connection to the API
           HttpURLConnection connection = (HttpURLConnection) new
URL(apiUrl).openConnection();

           try {
               // Using Scanner to read the API response
               Scanner scanner = new Scanner(connection.getInputStream());
               scanner.useDelimiter("\\A");
               String jsonStr = scanner.hasNext() ? scanner.next() : "";

               // Using Gson to parse JSON response
               Gson gson = new Gson();
               JsonObject jsonResponse = gson.fromJson(jsonStr,
JsonObject.class);
               JsonArray meals = jsonResponse.getAsJsonArray("meals");
```



```
                // Calculating API response time
                long endTime = System.currentTimeMillis();
                long apiResponseTime = endTime - startTime;
                // Calculating the size of the response in bytes
                int responseSize = jsonStr.getBytes("UTF-8").length;

                // Checking if any meals are found in the response
                if (meals != null && meals.size() > 0) {

                    // Extracting the first meal data
                    JsonObject mealData = meals.get(0).getAsJsonObject();
                    // Simplifying the response structure
                    JsonObject simplifiedResponse = new JsonObject();

                    // Adding picture and recipe properties to the simplified
response
                    simplifiedResponse.addProperty("picture",
mealData.get("strMealThumb").getAsString());
                    simplifiedResponse.addProperty("recipe",
mealData.get("strInstructions").getAsString());

                    // Collecting ingredients into a JSON array
                    JsonArray ingredients = new JsonArray();
                    for (int i = 1; i <= 20; i++) {
                        if (!mealData.get("strIngredient" + i).isJsonNull()) {
                            String ingredient = mealData.get("strIngredient" +
i).getAsString();
                            if (ingredient != null && !ingredient.isEmpty()) {
                                ingredients.add(ingredient);
                            }
                        }
                    }
                    simplifiedResponse.add("ingredients", ingredients);
                    // Logging successful request to MongoDB
                    logToMongo(request, mealName, startTime, apiResponseTime,
true, "", responseSize);
                    // Sending the simplified JSON response
                    response.getWriter().print(gson.toJson(simplifiedResponse));
                } else {
                    // Log including the responseSize for no meal data found
                    logToMongo(request, mealName, startTime, apiResponseTime,
false, "No meal data found for the given name.", responseSize);
                    response.getWriter().print("{\"error\": \"No meal data found
for the given name.\"}");
                }
                scanner.close();
            } catch (Exception e) {
                // Error handling, assuming minimal response size for error
```



```
                int responseSize = e.getMessage().getBytes("UTF-8").length;
                logToMongo(request, mealName, startTime,
System.currentTimeMillis() - startTime, false, e.getMessage(), responseSize);
                e.printStackTrace();
                response.getWriter().print("{\"error\": \"There was an error
processing your request.\"}");
            } finally {
                connection.disconnect();
            }
        } else {
            // Handling the case where no meal name is provided
            int responseSize = "No meal name
provided.".getBytes("UTF-8").length;
            logToMongo(request, "", startTime, System.currentTimeMillis() -
startTime, false, "No meal name provided.", responseSize);
            response.getWriter().print("{\"error\": \"No meal name
provided.\"}");
        }
    }

    /**
     * Logs request and response details to MongoDB.
     *
     * @param request           The HTTP request object.
     * @param mealName          The name of the meal queried.
     * @param requestTimestamp The time when the request was initiated.
     * @param apiResponseTime  The time taken to get the response from the API.
     * @param isSuccess         Flag indicating if the API request was
successful.
     * @param errorMessage      The error message, if any.
     * @param responseSize      The size of the API response in bytes.
     */
    private void logToMongo(HttpServletRequest request, String mealName, long
requestTimestamp, long apiResponseTime, boolean isSuccess, String errorMessage,
int responseSize) {
        // Creating a document to log in MongoDB
        Document log = new Document("mealName", mealName)
                .append("requestTimestamp", requestTimestamp)
                .append("apiResponseTime", apiResponseTime)
                .append("userAgent", request.getHeader("User-Agent"))
                .append("status", isSuccess ? "Success" : "Error")
                .append("errorMessage", errorMessage)
                .append("responseSize", responseSize);
        // Inserting the log document into the MongoDB collection
        collection.insertOne(log);
    }
}
```

Requirement 4

1. Meal Name:
   - The `mealName` parameter from the mobile phone request is logged. This is crucial as it defines the user's search query and is central to the processing logic of the web service.

2. Request Timestamp:
   - The exact time when the request was received from the mobile phone is captured (`requestTimestamp`). This timestamp is essential for performance monitoring and understanding the service's usage patterns.

3. API Response Time:
   - The time taken by the third-party API to respond (`apiResponseTime`) is calculated by measuring the interval between the request initiation and the reception of the response. This information is vital for diagnosing any latency issues in the third-party API.

4. User Agent:
   - The `User-Agent` header from the HTTP request, which typically contains details about the mobile phone and operating system, is logged. This information can be used to analyze the diversity of devices accessing the service and tailor optimizations accordingly.

5. Status (Success/Error):
   - The success or error status of the interaction is logged. If the request is successful and meals data is fetched, it logs as "Success"; otherwise, it logs as "Error", providing a straightforward way to track the reliability and functioning of the web service.

6. Response Size:
   - The size of the response sent back to the mobile phone (`responseSize`) is recorded in bytes. This metric is useful for monitoring the amount of data transmitted, which can impact mobile data usage and performance.

7. Error Message (if any):
   - If the process encounters an error, the error message is logged (`errorMessage`). This inclusion is critical for debugging and improving the web service by understanding what goes wrong during failures.

Requirement 5

The `MealServlet` class within the web service architecture demonstrates effective interaction with a MongoDB database hosted in the cloud, fulfilling the requirements of connecting to, storing, and retrieving log information from a database. Here's how this is accomplished:

Connection to MongoDB

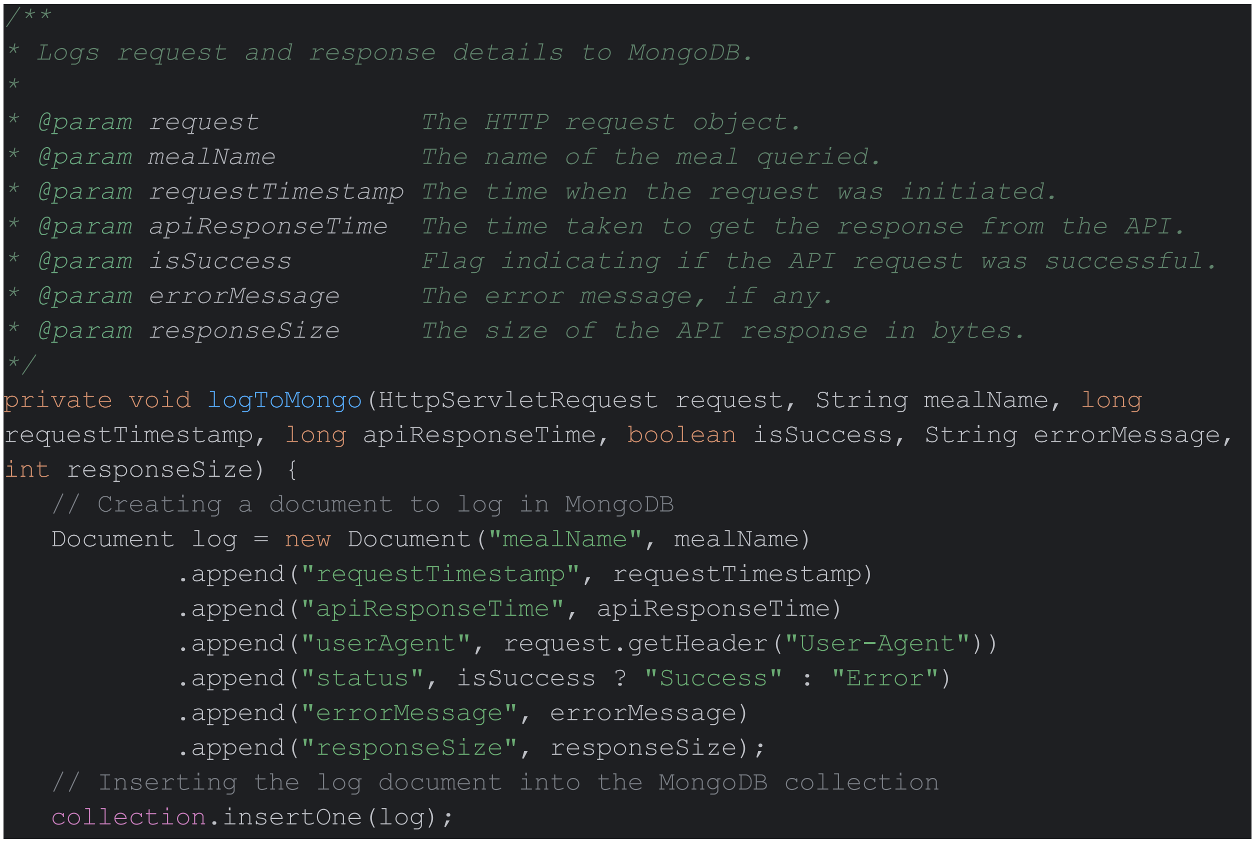
1. Initialization:
   - The servlet initializes a connection to the MongoDB database in the constructor. It uses a connection string that includes the database credentials (`psingh3:Priyanshi23`) and the database address (`cluster0.7yt1mug.mongodb.net`). This string specifies the MongoDB cluster hosted on MongoDB Atlas, which is a fully-managed cloud database.

2. Database and Collection Access:
   - The servlet accesses a specific database (`Project4`) and selects the `MealLogs` collection within this database to store log entries. The `MongoDatabase` and `MongoCollection<Document>` classes from the MongoDB Java driver are utilized to manage these operations seamlessly.

Storing Log Information

1. Document Creation:
   - When a request is processed, the servlet creates a `Document` object to store structured log data. This document includes several pieces of information such as the meal name, timestamp, API response time, user agent, success status, error message (if any), and the response size.

2. Insert Operation:
   - The created `Document` is inserted into the `MealLogs` collection using the `insertOne()` method. This operation adds the log entry to the database, where it is stored persistently.



```
/**
* Logs request and response details to MongoDB.
*
* @param request          The HTTP request object.
* @param mealName         The name of the meal queried.
* @param requestTimestamp The time when the request was initiated.
* @param apiResponseTime  The time taken to get the response from the API.
* @param isSuccess        Flag indicating if the API request was successful.
* @param errorMessage     The error message, if any.
* @param responseSize     The size of the API response in bytes.
*/
private void logToMongo(HttpServletRequest request, String mealName, long
requestTimestamp, long apiResponseTime, boolean isSuccess, String errorMessage,
int responseSize) {
   // Creating a document to log in MongoDB
   Document log = new Document("mealName", mealName)
           .append("requestTimestamp", requestTimestamp)
           .append("apiResponseTime", apiResponseTime)
           .append("userAgent", request.getHeader("User-Agent"))
           .append("status", isSuccess ? "Success" : "Error")
           .append("errorMessage", errorMessage)
           .append("responseSize", responseSize);
   // Inserting the log document into the MongoDB collection
   collection.insertOne(log);
}
```

Requirement 6

The provided solution effectively addresses the requirement of displaying operations analytics and full logs on a web-based dashboard, which is accessible via a unique URL (`/analytics`). Here’s how each part of the requirement has been fulfilled:

A. Unique URL Addressing the Web Interface Dashboard
- The `AnalyticsServlet` is mapped to the URL pattern `/analytics` using the `@WebServlet` annotation. This makes it accessible via a unique URL that points directly to the analytics dashboard interface.

B. Displaying at Least Three Interesting Operations Analytics
1. Top 10 Search Terms: This analytic shows the most frequently searched meal names, providing insights into user preferences or popular trends.
2. Average API Response Latency: This metric indicates the average time taken by the third-party API to respond to requests, which is crucial for monitoring the performance and efficiency of external integrations.
3. Most Popular Phone Models: By analyzing the user agents from the logs, this analytic displays the most commonly used devices accessing the service, helping understand the technology landscape of the user base.

C. Displaying Formatted Full Logs
- The dashboard includes a detailed log table displaying comprehensive information for each logged entry, such as:
  - Timestamp: The exact time when the request was made.
  - Meal Name: The meal searched for, indicating user interest.
  - User Agent: Identifies the device making the request, useful for technical insights and optimization.
  - API Response Time: How long the third-party API took to respond, relevant for performance analysis.
  - Status: Indicates whether the request was processed successfully or resulted in an error.
  - Response Size: The amount of data sent back to the client, pertinent for assessing payload efficiency.
  -Error Message: Details any errors encountered during processing, crucial for debugging and service improvement.

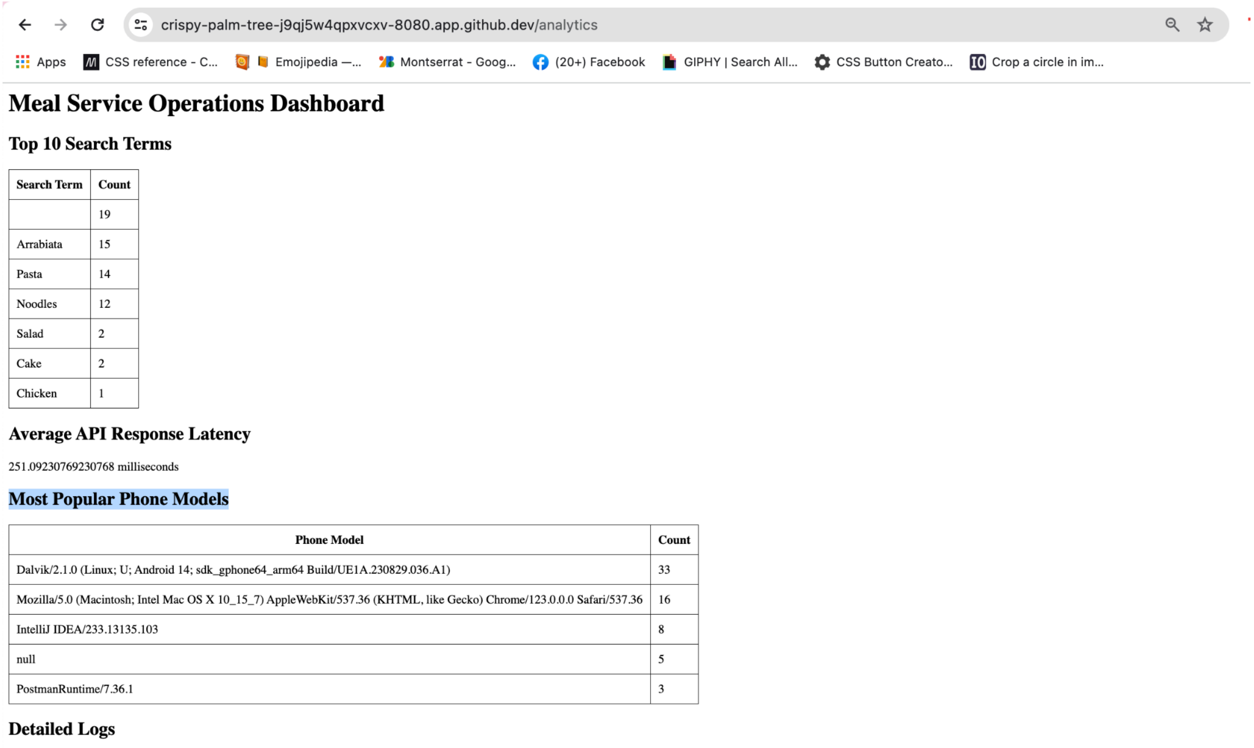
Implementation Details:

Servlet Backend
The `AnalyticsServlet` handles the aggregation of log data from the MongoDB database and prepares it for display. It uses MongoDB’s aggregation pipeline to compute the analytics, making

effective use of operations like `group`, `sum`, `avg`, and `sort` to extract meaningful statistics from the collected data.

JSP Frontend

The JSP (`dashboard.jsp`) serves as the presentation layer. It uses standard HTML tables styled with CSS to present the data in an organized and readable format. Server-side scripting in JSP fetches the data attributes set by the servlet and iterates over them to render the tables dynamically.



Meal Service Operations Dashboard

Top 10 Search Terms

Search Term	Count
	19
Arrabiata	15
Pasta	14
Noodles	12
Salad	2
Cake	2
Chicken	1

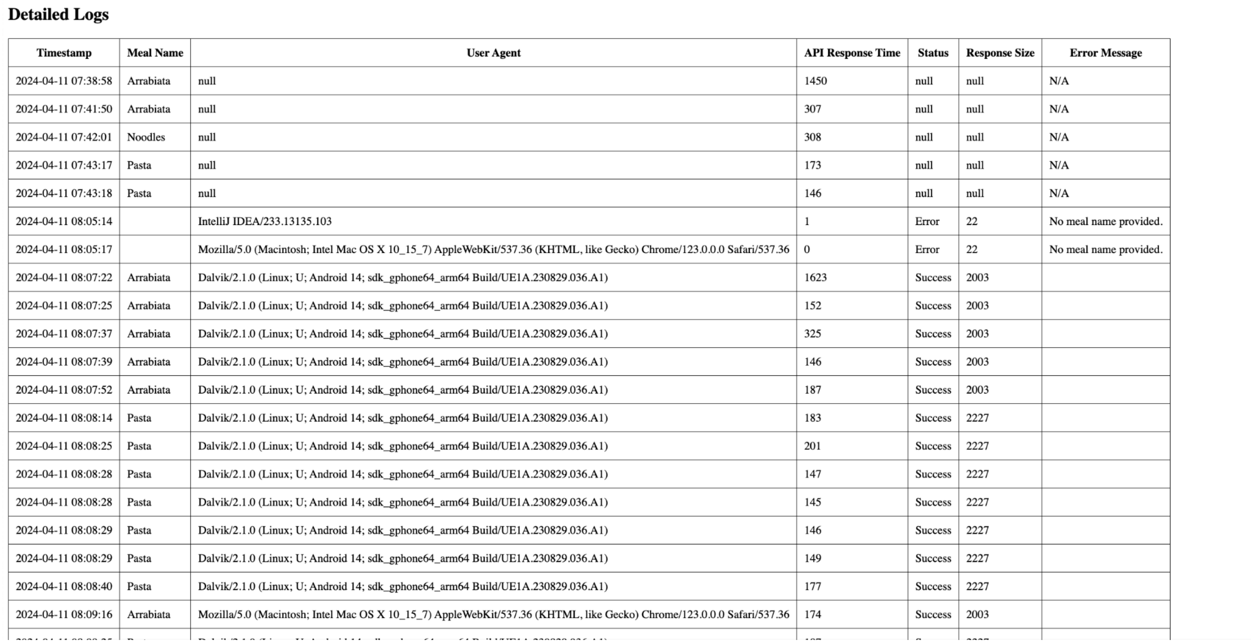
Average API Response Latency

251.09230769230768 milliseconds

Most Popular Phone Models

Phone Model	Count
Dalvik/2.1.0 (Linux; U; Android 14; sdk_gphone64_arm64 Build/UE1A.230829.036.A1)	33
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/123.0.0.0 Safari/537.36	16
IntelliJ IDEA/233.13135.103	8
null	5
PostmanRuntime/7.36.1	3

Detailed Logs

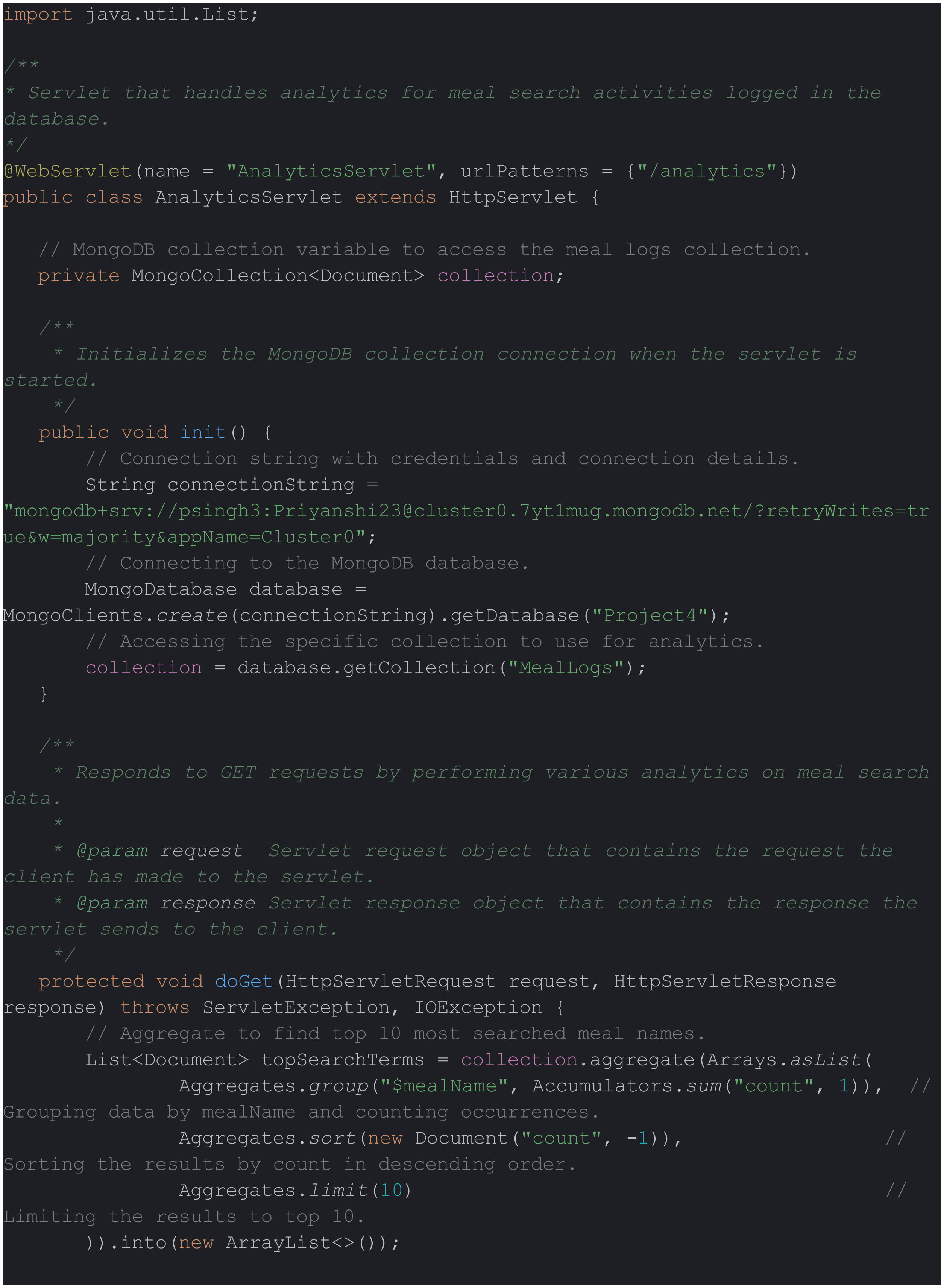


Detailed Logs

Timestamp	Meal Name	User Agent	API Response Time	Status	Response Size	Error Message
2024-04-11 07:38:58	Arrabiata	null	1450	null	null	N/A
2024-04-11 07:41:50	Arrabiata	null	307	null	null	N/A
2024-04-11 07:42:01	Noodles	null	308	null	null	N/A
2024-04-11 07:43:17	Pasta	null	173	null	null	N/A
2024-04-11 07:43:18	Pasta	null	146	null	null	N/A
2024-04-11 08:05:14		IntelliJ IDEA/233.13135.103	1	Error	22	No meal name provided.
2024-04-11 08:05:17		Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/123.0.0.0 Safari/537.36	0	Error	22	No meal name provided.
2024-04-11 08:07:22	Arrabiata	Dalvik/2.1.0 (Linux; U; Android 14; sdk_gphone64_arm64 Build/UE1A.230829.036.A1)	1623	Success	2003	
2024-04-11 08:07:25	Arrabiata	Dalvik/2.1.0 (Linux; U; Android 14; sdk_gphone64_arm64 Build/UE1A.230829.036.A1)	152	Success	2003	
2024-04-11 08:07:37	Arrabiata	Dalvik/2.1.0 (Linux; U; Android 14; sdk_gphone64_arm64 Build/UE1A.230829.036.A1)	325	Success	2003	
2024-04-11 08:07:39	Arrabiata	Dalvik/2.1.0 (Linux; U; Android 14; sdk_gphone64_arm64 Build/UE1A.230829.036.A1)	146	Success	2003	
2024-04-11 08:07:52	Arrabiata	Dalvik/2.1.0 (Linux; U; Android 14; sdk_gphone64_arm64 Build/UE1A.230829.036.A1)	187	Success	2003	
2024-04-11 08:08:14	Pasta	Dalvik/2.1.0 (Linux; U; Android 14; sdk_gphone64_arm64 Build/UE1A.230829.036.A1)	183	Success	2227	
2024-04-11 08:08:25	Pasta	Dalvik/2.1.0 (Linux; U; Android 14; sdk_gphone64_arm64 Build/UE1A.230829.036.A1)	201	Success	2227	
2024-04-11 08:08:28	Pasta	Dalvik/2.1.0 (Linux; U; Android 14; sdk_gphone64_arm64 Build/UE1A.230829.036.A1)	147	Success	2227	
2024-04-11 08:08:28	Pasta	Dalvik/2.1.0 (Linux; U; Android 14; sdk_gphone64_arm64 Build/UE1A.230829.036.A1)	145	Success	2227	
2024-04-11 08:08:29	Pasta	Dalvik/2.1.0 (Linux; U; Android 14; sdk_gphone64_arm64 Build/UE1A.230829.036.A1)	146	Success	2227	
2024-04-11 08:08:29	Pasta	Dalvik/2.1.0 (Linux; U; Android 14; sdk_gphone64_arm64 Build/UE1A.230829.036.A1)	149	Success	2227	
2024-04-11 08:08:40	Pasta	Dalvik/2.1.0 (Linux; U; Android 14; sdk_gphone64_arm64 Build/UE1A.230829.036.A1)	177	Success	2227	
2024-04-11 08:09:16	Arrabiata	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/123.0.0.0 Safari/537.36	174	Success	2003	

AnalyticsServlet.java

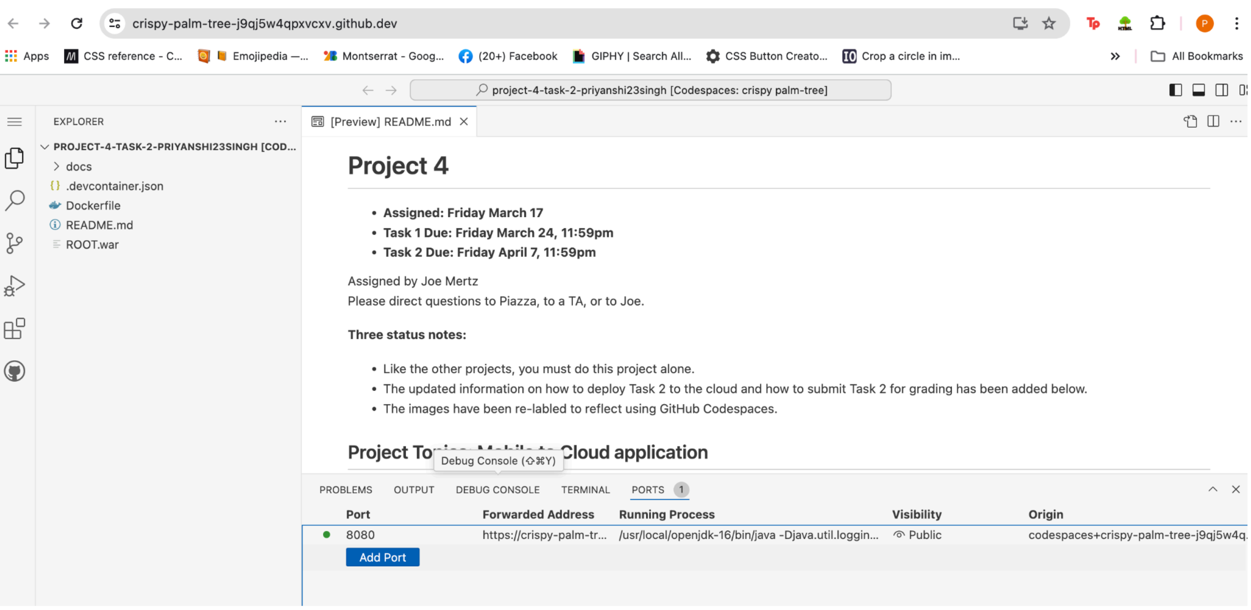
```
/**
* @author: Priyanshi Singh
* psingh3
*/
package ds.proj4;

// Importing required MongoDB client classes for database interactions.
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
// Importing aggregation framework classes for performing complex queries and
aggregations.
import com.mongodb.client.model.Accumulators;
import com.mongodb.client.model.Aggregates;
// Importing servlet classes to define servlet functionalities.
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
// Importing BSON document class for handling BSON format, used in MongoDB.
import org.bson.Document;

// Importing Java utility classes.
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
```



```
import java.util.List;

/**
* Servlet that handles analytics for meal search activities logged in the
database.
*/
@WebServlet(name = "AnalyticsServlet", urlPatterns = {"/analytics"})
public class AnalyticsServlet extends HttpServlet {

   // MongoDB collection variable to access the meal logs collection.
   private MongoCollection<Document> collection;

   /**
    * Initializes the MongoDB collection connection when the servlet is
started.
    */
   public void init() {
       // Connection string with credentials and connection details.
       String connectionString =
"mongodb+srv://psingh3:Priyanshi23@cluster0.7yt1mug.mongodb.net/?retryWrites=tr
ue&w=majority&appName=Cluster0";
       // Connecting to the MongoDB database.
       MongoDatabase database =
MongoClients.create(connectionString).getDatabase("Project4");
       // Accessing the specific collection to use for analytics.
       collection = database.getCollection("MealLogs");
   }

   /**
    * Responds to GET requests by performing various analytics on meal search
data.
    *
    * @param request  Servlet request object that contains the request the
client has made to the servlet.
    * @param response Servlet response object that contains the response the
servlet sends to the client.
    */
   protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
       // Aggregate to find top 10 most searched meal names.
       List<Document> topSearchTerms = collection.aggregate(Arrays.asList(
               Aggregates.group("$mealName", Accumulators.sum("count", 1)),  //
Grouping data by mealName and counting occurrences.
               Aggregates.sort(new Document("count", -1)),                  //
Sorting the results by count in descending order.
               Aggregates.limit(10)                                         //
Limiting the results to top 10.
       )).into(new ArrayList<>());
```

```
        // Calculate the average response time from the API.
        Document avgApiResponseTime = collection.aggregate(Arrays.asList(
                Aggregates.group(null, Accumulators.avg("avgApiResponseTime",
"$apiResponseTime"))  // Calculating average of API response times.
        )).first();  // Getting the first document of the aggregation result.

        // Aggregate to find the top 5 most popular user agents (phone models).
        List<Document> popularPhoneModels = collection.aggregate(Arrays.asList(
                Aggregates.group("$userAgent", Accumulators.sum("count", 1)),
// Grouping by userAgent and counting occurrences.
                Aggregates.sort(new Document("count", -1)),                   //
Sorting by count in descending order.
                Aggregates.limit(5)                                           //
Limiting the results to top 5.
        )).into(new ArrayList<>());

        // Set attributes for forwarding to views (JSP pages).
        request.setAttribute("topSearchTerms", topSearchTerms);
        request.setAttribute("avgApiResponseTime", avgApiResponseTime != null ?
avgApiResponseTime.getDouble("avgApiResponseTime") : 0.0);
        request.setAttribute("popularPhoneModels", popularPhoneModels);

        // Retrieve the latest 100 detailed log entries from the collection.
        List<Document> detailedLogs = collection.find().limit(100).into(new
ArrayList<>());  // Example: limit to the last 100 logs.
        request.setAttribute("detailedLogs", detailedLogs);

        // Forward the request to the dashboard.jsp page to display the
analytics.
        request.getRequestDispatcher("/dashboard.jsp").forward(request,
response);
    }
}
```

Requirement 7:



Project 4

- **Assigned: Friday March 17**
- **Task 1 Due: Friday March 24, 11:59pm**
- **Task 2 Due: Friday April 7, 11:59pm**

Assigned by Joe Mertz
Please direct questions to Piazza, to a TA, or to Joe.

Three status notes:

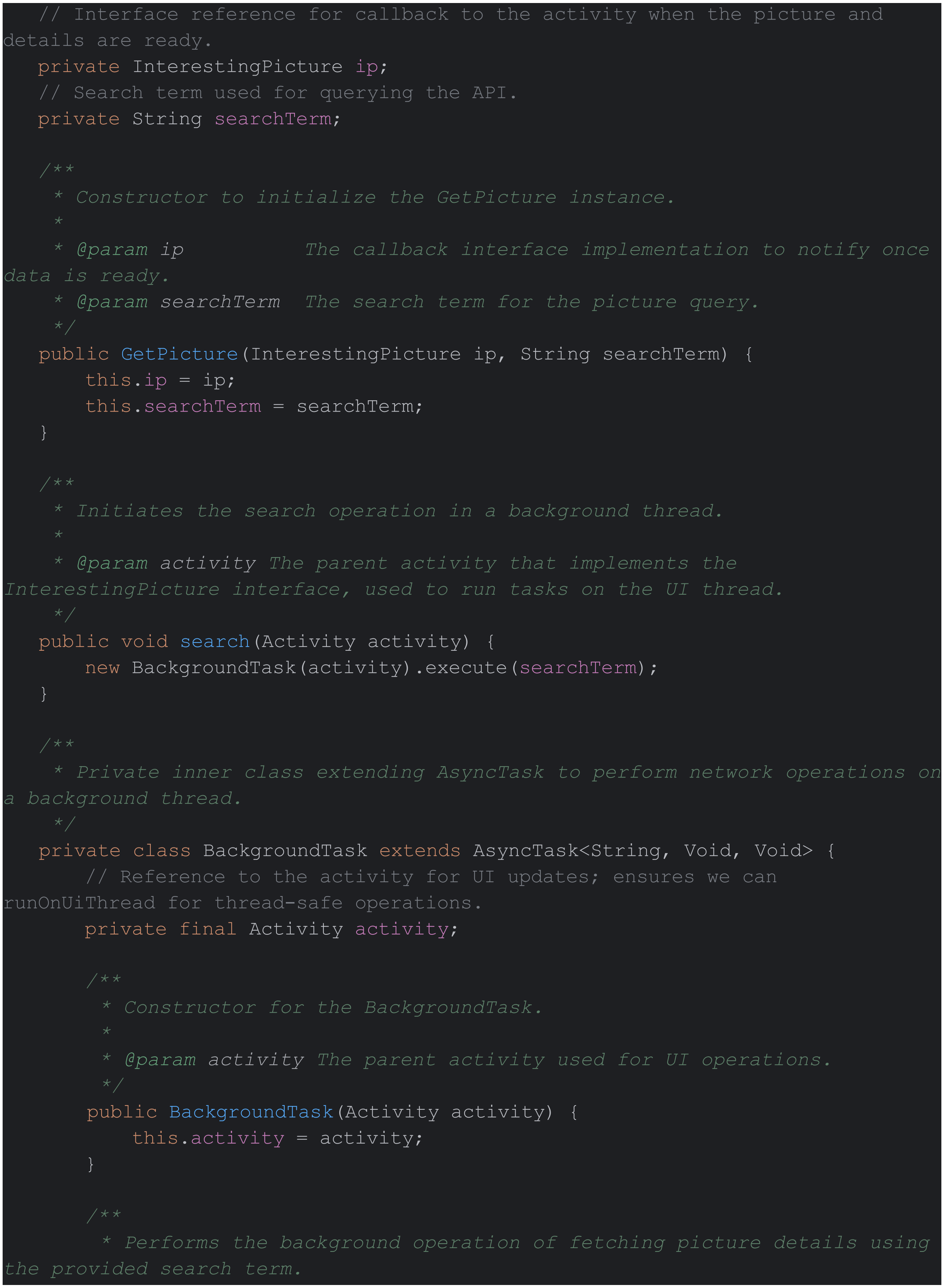
- Like the other projects, you must do this project alone.
- The updated information on how to deploy Task 2 to the cloud and how to submit Task 2 for grading has been added below.
- The images have been re-labled to reflect using GitHub Codespaces.

Project To... Cloud application

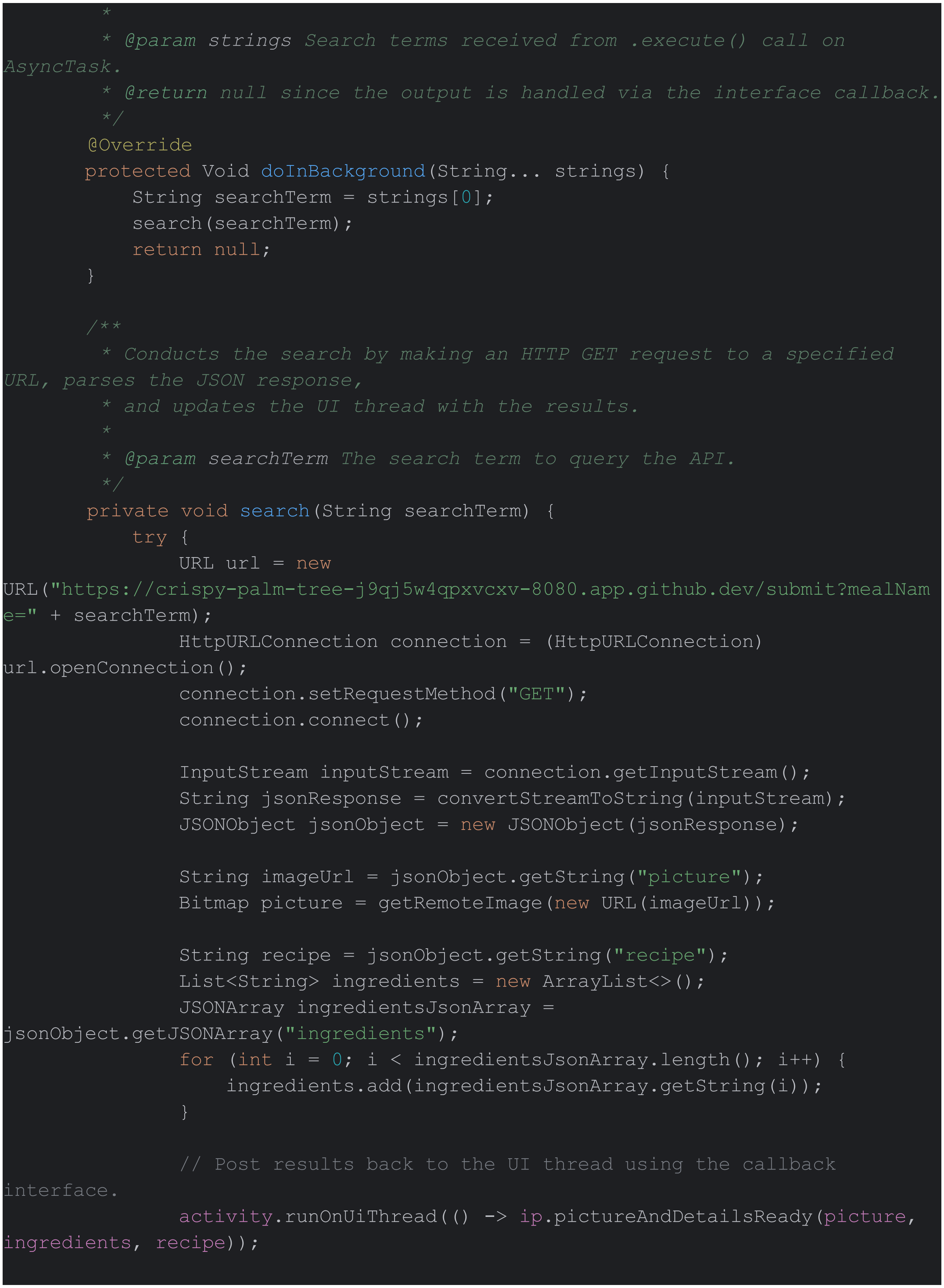
	Port	Forwarded Address	Running Process	Visibility	Origin
●	8080	https://crispy-palm-tr...	/usr/local/openjdk-16/bin/java -Djava.util.loggin...	Public	codespaces+crispy-palm-tree-j9qj5w4q.

GetPicture.java

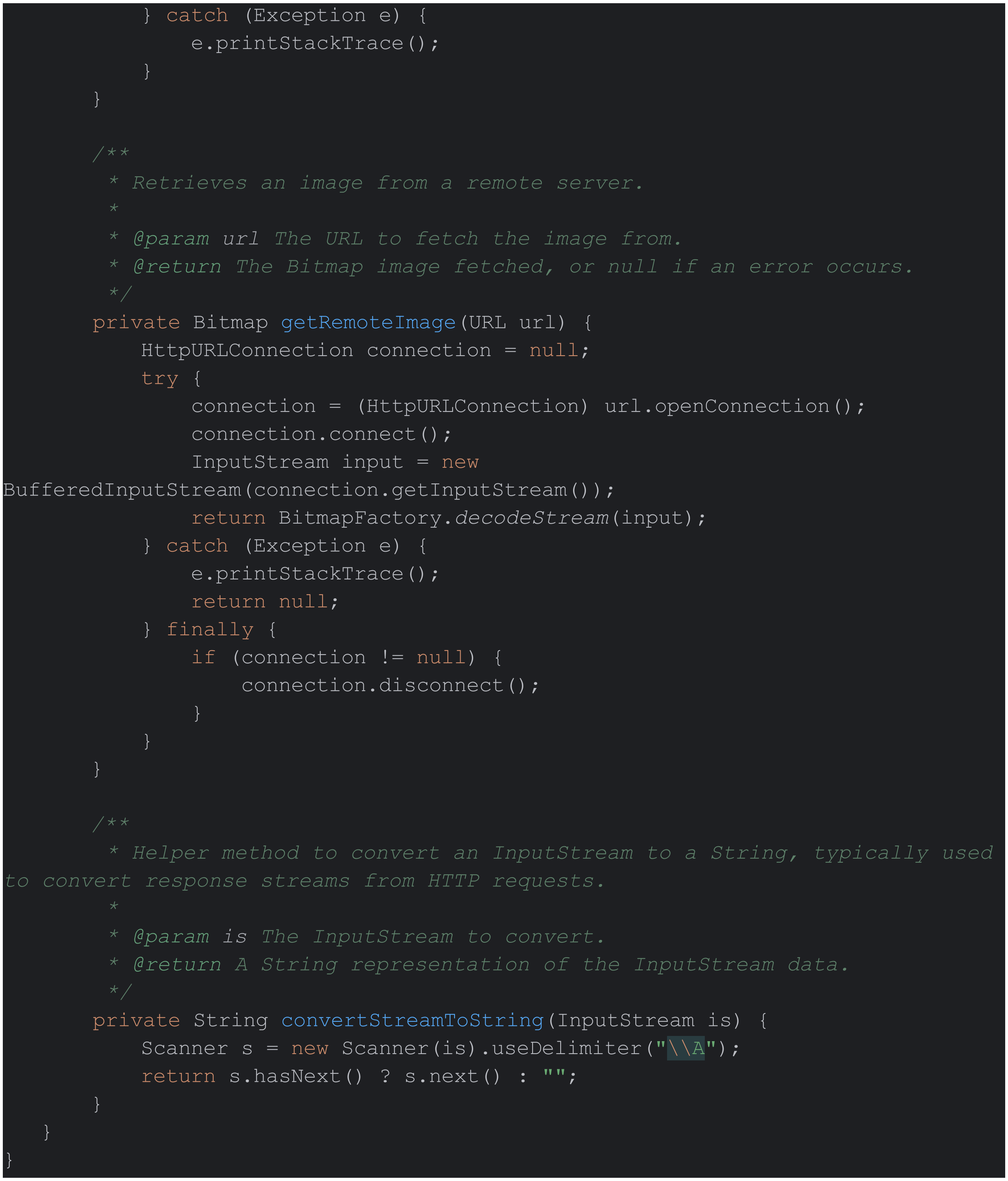
```
/**
* @author Priyanshi Singh
* psingh3
*/
package com.example.dspicture;

// Importing necessary Android, networking, and JSON handling classes.
import android.app.Activity;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.os.AsyncTask;
import org.json.JSONArray;
import org.json.JSONObject;

import java.io.BufferedInputStream;
import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

/**
* This class handles retrieving a picture and its details based on a search
term using an API.
*/
public class GetPicture {
```



```
    // Interface reference for callback to the activity when the picture and
details are ready.
    private InterestingPicture ip;
    // Search term used for querying the API.
    private String searchTerm;

    /**
     * Constructor to initialize the GetPicture instance.
     *
     * @param ip           The callback interface implementation to notify once
data is ready.
     * @param searchTerm  The search term for the picture query.
     */
    public GetPicture(InterestingPicture ip, String searchTerm) {
        this.ip = ip;
        this.searchTerm = searchTerm;
    }

    /**
     * Initiates the search operation in a background thread.
     *
     * @param activity The parent activity that implements the
InterestingPicture interface, used to run tasks on the UI thread.
     */
    public void search(Activity activity) {
        new BackgroundTask(activity).execute(searchTerm);
    }

    /**
     * Private inner class extending AsyncTask to perform network operations on
a background thread.
     */
    private class BackgroundTask extends AsyncTask<String, Void, Void> {
        // Reference to the activity for UI updates; ensures we can
runOnUiThread for thread-safe operations.
        private final Activity activity;

        /**
         * Constructor for the BackgroundTask.
         *
         * @param activity The parent activity used for UI operations.
         */
        public BackgroundTask(Activity activity) {
            this.activity = activity;
        }

        /**
         * Performs the background operation of fetching picture details using
the provided search term.
```



```
         *
         * @param strings Search terms received from .execute() call on
AsyncTask.
         * @return null since the output is handled via the interface callback.
         */
        @Override
        protected Void doInBackground(String... strings) {
            String searchTerm = strings[0];
            search(searchTerm);
            return null;
        }

        /**
         * Conducts the search by making an HTTP GET request to a specified
URL, parses the JSON response,
         * and updates the UI thread with the results.
         *
         * @param searchTerm The search term to query the API.
         */
        private void search(String searchTerm) {
            try {
                URL url = new
URL("https://crispy-palm-tree-j9qj5w4qpxvcxv-8080.app.github.dev/submit?mealNam
e=" + searchTerm);
                HttpURLConnection connection = (HttpURLConnection)
url.openConnection();
                connection.setRequestMethod("GET");
                connection.connect();

                InputStream inputStream = connection.getInputStream();
                String jsonResponse = convertStreamToString(inputStream);
                JSONObject jsonObject = new JSONObject(jsonResponse);

                String imageUrl = jsonObject.getString("picture");
                Bitmap picture = getRemoteImage(new URL(imageUrl));

                String recipe = jsonObject.getString("recipe");
                List<String> ingredients = new ArrayList<>();
                JSONArray ingredientsJsonArray =
jsonObject.getJSONArray("ingredients");
                for (int i = 0; i < ingredientsJsonArray.length(); i++) {
                    ingredients.add(ingredientsJsonArray.getString(i));
                }

                // Post results back to the UI thread using the callback
interface.
                activity.runOnUiThread(() -> ip.pictureAndDetailsReady(picture,
ingredients, recipe));
```



```
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        /**
         * Retrieves an image from a remote server.
         *
         * @param url The URL to fetch the image from.
         * @return The Bitmap image fetched, or null if an error occurs.
         */
        private Bitmap getRemoteImage(URL url) {
            HttpURLConnection connection = null;
            try {
                connection = (HttpURLConnection) url.openConnection();
                connection.connect();
                InputStream input = new
BufferedInputStream(connection.getInputStream());
                return BitmapFactory.decodeStream(input);
            } catch (Exception e) {
                e.printStackTrace();
                return null;
            } finally {
                if (connection != null) {
                    connection.disconnect();
                }
            }
        }

        /**
         * Helper method to convert an InputStream to a String, typically used
to convert response streams from HTTP requests.
         *
         * @param is The InputStream to convert.
         * @return A String representation of the InputStream data.
         */
        private String convertStreamToString(InputStream is) {
            Scanner s = new Scanner(is).useDelimiter("\\A");
            return s.hasNext() ? s.next() : "";
        }
    }
}
```