

Obiajuru Triumph Nwadiokwu
AndrewID: OTN
Implement a native Android application
1a

Our application's layout includes several different kinds of Views, each extending `android.view.View`:

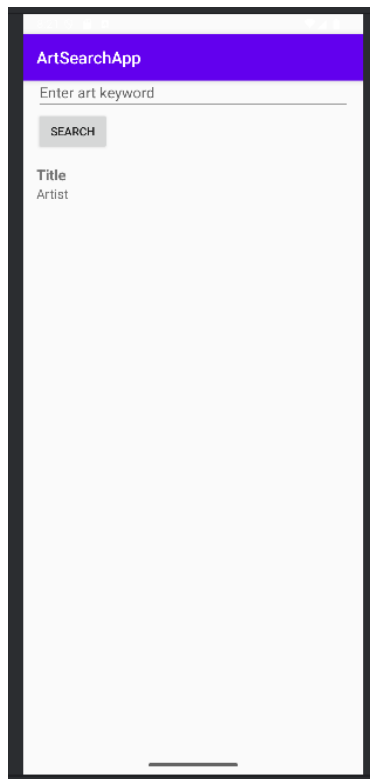
`EditText`: Allows the user to input a search query (e.g., "monet").

`TextView`: Displays the artwork title and artist information.

`ImageView`: Shows the artwork image retrieved from the web service.

`Button`: Triggers the search when pressed.

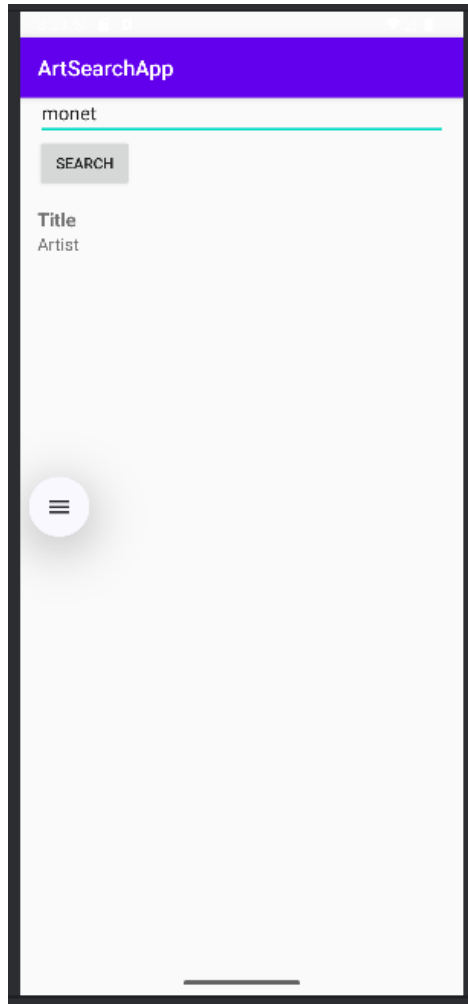
Together, these components ensure that the UI uses at least three different types of views.



b.

Requires Input from the User:

The app uses an EditText where users enter their search term. This input is essential as it drives the search functionality for retrieving specific artworks from the web service.



c. Makes an HTTP Request in a Background Thread:

When the user taps the search button, the app initiates an HTTP GET request to the web service endpoint using a background thread. This is implemented with an **ExecutorService** ensuring the network operation does not block the main (UI) thread.

// Define an ExecutorService to run background tasks

private final ExecutorService executor = Executors.newSingleThreadExecutor();

// Create a Handler associated with the main thread to update the UI later

private final Handler handler = new Handler(Looper.getMainLooper());

private void searchArtwork(String query) {

executor.execute() -> {

try {

// Build the URL including the query parameter

URL url = new URL(servletUrl + "?query=" + query);

HttpURLConnection connection = (HttpURLConnection) url.openConnection();

connection.setRequestMethod("GET");

connection.setConnectTimeout(5000); // Connection timeout

connection.setReadTimeout(5000); // Read timeout

// Check for a successful response

int responseCode = connection.getResponseCode();

if (responseCode != HttpURLConnection.HTTP_OK) {

throw new Exception("Server returned error: " + responseCode);

}

// Read the response stream

BufferedReader in = new BufferedReader(new

InputStreamReader(connection.getInputStream()));

StringBuilder responseBuilder = new StringBuilder();

String line;

while ((line = in.readLine()) != null) {

responseBuilder.append(line);

}

in.close();

```

// Parse the JSON response
JSONObject json = new JSONObject(responseBuilder.toString());
// The response is structured as: { "artworks": [ { ... }, ... ] }
JSONArray artworks = json.optJSONArray("artworks");

// Default values if no artwork is found
String title = "N/A";
String artist = "Unknown";
String imageUrl = "";

if (artworks != null && artworks.length() > 0) {
    // Use the first artwork from the array
    JSONObject firstArtwork = artworks.getJSONObject(0);
    title = firstArtwork.optString("title", "N/A");
    artist = firstArtwork.optString("artist_display", "Unknown");
    imageUrl = firstArtwork.optString("imageUrl", "");
}

// Post updates back to the main thread
final String finalTitle = title;
final String finalArtist = artist;
final String finalImageUrl = imageUrl;
handler.post() -> {
    textViewTitle.setText("Title: " + finalTitle);
    textViewArtist.setText("Artist: " + finalArtist);

    if (!finalImageUrl.isEmpty()) {
        // Load the artwork image using Glide
        Glide.with(MainActivity.this)
            .load(finalImageUrl)
            .into(imageViewArt);
    } else {
        // Set a fallback image if no image URL is provided
        imageViewArt.setImageResource(R.drawable.ic_launcher_foreground);
    }
});
} catch (Exception e) {
    e.printStackTrace();
    handler.post() -> {

```

```

        Toast.makeText(MainActivity.this, "Error: " + e.getMessage(),
Toast.LENGTH_LONG).show();
        textViewTitle.setText("Title: N/A");
        textViewArtist.setText("Artist: Unknown");
        imageViewArt.setImageResource(R.drawable.ic_launcher_foreground);
    });
}
});
}
}

```

Key Points Explained:

ExecutorService and Background Thread:

The `executor.execute()` call runs the HTTP request and JSON parsing on a background thread, ensuring that these potentially long-running operations do not block the main UI thread.

Handler for UI Updates:

The `handler.post()` method is used to safely update the UI (such as setting text in TextViews and loading an image in an ImageView) on the main thread after the background processing is complete.

Repeatable Operations:

The method `searchArtwork()` can be called multiple times—each time the user submits a search query, a new background task is created, making the app fully repeatable without restarting.

This section of the code clearly satisfies requirement 1(c) by demonstrating:

The use of a background thread for network operations.

The correct parsing of a JSON formatted reply.

The safe update of UI components using a Handler

1d

In the `searchArtwork(String query)` method of `MainActivity`, the application reads the HTTP response from your servlet and converts the returned string into a JSON object. Here is the relevant part of your code, with commentary

```
// Read the response stream
    BufferedReader in = new BufferedReader(new
InputStreamReader(connection.getInputStream()));
    StringBuilder responseBuilder = new StringBuilder();
    String line;
    while ((line = in.readLine()) != null) {
        responseBuilder.append(line);
    }
    in.close();

    // Parse the JSON response
    JSONObject json = new JSONObject(responseBuilder.toString());
    // The response is structured as: { "artworks": [ { ... }, ... ] }
    JSONArray artworks = json.optJSONArray("artworks");

    // Default values if no artwork is found
    String title = "N/A";
    String artist = "Unknown";
    String imageUrl = "";

    if (artworks != null && artworks.length() > 0) {
        // Use the first artwork from the array
        JSONObject firstArtwork = artworks.getJSONObject(0);
        title = firstArtwork.optString("title", "N/A");
        artist = firstArtwork.optString("artist_display", "Unknown");
        imageUrl = firstArtwork.optString("imageUrl", "");
    }
```

1. Reading the Response:

- A `BufferedReader` is used to read the incoming data stream from the web service line by line. The data is appended to a `StringBuilder` so we end up with a

complete JSON string.

2. Creating a JSON Object:

The raw string is passed to the JSONObject constructor:

```
JSONObject json = new JSONObject(responseBuilder.toString());
```

- This step converts the textual JSON data into a structured JSONObject that can be easily navigated.

3. Navigating the JSON Structure:

The "artworks" key in the top-level JSON object contains an array of artworks. We retrieve it with:

```
JSONArray artworks = json.optJSONArray("artworks");
```

-
- If this array is present and non-empty, we extract details from the first artwork in the array (for demonstration purposes).

4. Extracting Fields:

- Fields such as "title", "artist_display", and "imageUrl" are pulled out using the optString(...) method. This allows the code to handle missing or null values gracefully by providing a default value (e.g., "N/A").

5. Displaying the Parsed Data:

- Finally, the parsed information (title, artist, image URL) is displayed on the UI (e.g., in a TextView or an ImageView) once the background thread processing is complete.

In short, the code reads the web service's JSON reply via an HTTP connection, constructs a JSONObject from it, and then extracts the necessary fields. This fully satisfies the requirement to receive and parse a JSON-formatted reply from your web service.

1e

```
// Post updates back to the main thread
    final String finalTitle = title;
    final String finalArtist = artist;
    final String finalImageUrl = imageUrl;
    handler.post() -> {
        textViewTitle.setText("Title: " + finalTitle);
        textViewArtist.setText("Artist: " + finalArtist);

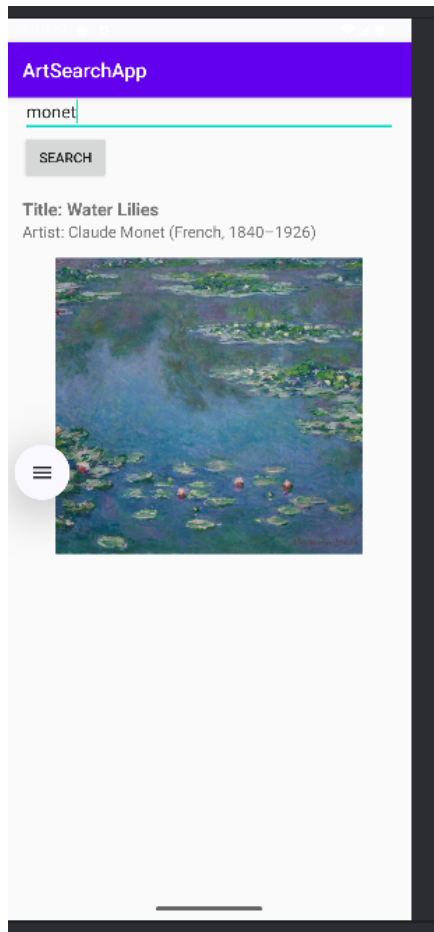
        if (!finalImageUrl.isEmpty()) {
            // Load the artwork image using Glide
            Glide.with(MainActivity.this)
                .load(finalImageUrl)
                .into(imageViewArt);
        } else {
            // Set a fallback image if no image URL is provided
            imageViewArt.setImageResource(R.drawable.ic_launcher_foreground);
        }
    });
} catch (Exception e) {
    e.printStackTrace();
    handler.post() -> {
        Toast.makeText(MainActivity.this, "Error: " + e.getMessage(),
Toast.LENGTH_LONG).show();
        textViewTitle.setText("Title: N/A");
        textViewArtist.setText("Artist: Unknown");
        imageViewArt.setImageResource(R.drawable.ic_launcher_foreground);
    });
}
});
}
```

TextViews and ImageView Update: The application immediately displays the new artwork's title and artist in the respective TextViews, and the ImageView is updated to show the new artwork image.

Seamless User Experience: Because these updates occur on the main UI thread (via the Handler), users see the latest information almost instantly after the search is completed.

Repeatable Action: Users can perform multiple searches, and the UI will update each time with the corresponding new results without needing to restart the application.

This process fulfills Requirement 1(e) by ensuring that every new search query results in a refreshed display of data, thus providing a responsive and dynamic user interface.



Our Android application is designed so that the user can perform multiple searches consecutively without needing to restart the application. Here's how we ensure repeatability:

Reentrant Search Functionality:

Each time the user taps the search button, the `searchArtwork(String query)` method is invoked. This method:

Reads the current search query from the `EditText`.

Executes a new HTTP request on a background thread.

Parses the JSON response and updates the UI with the new data.

This process can be repeated as many times as needed.

Stateless UI Update:

The UI components (`EditText`, `TextViews`, and `ImageView`) are updated with each new search. There is no persistent state that prevents the UI from being refreshed; each button click triggers a full cycle of fetching and displaying data.

No Application Restart Required:

Since the application's code does not lock or disable the search functionality after one use, users can continuously enter new search terms, trigger the background request, and receive new results. The design ensures that any previous data is overwritten or replaced with the new search results.

Robust Error Handling:

Even if a network error or unexpected response occurs, error messages are displayed and the UI remains responsive, allowing the user to try another search without restarting the app.

Example from the Code:

Every time the search button is clicked, the `searchArtwork(query)` method is called:

```
buttonSearch.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        String query = editTextQuery.getText().toString().trim();
```

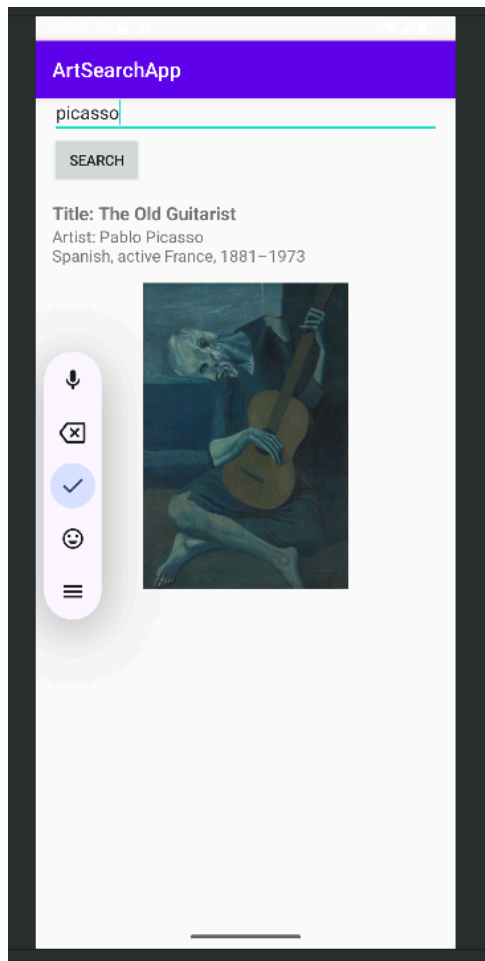
```

        // Input validation
        if (query.isEmpty()) {
            Toast.makeText(MainActivity.this, "Please enter a search term",
Toast.LENGTH_SHORT).show();
            return;
        }

        // Initiate background search with the user query
        searchArtwork(query);
    }
});
}

```

This mechanism confirms that our application is repeatable, fulfilling Requirement 1(f) as users can continually search and refresh data without restarting the app.



Implementation 2

2a

Implement a Simple API

Explanation:

We have implemented a simple, single-path API that listens for HTTP GET requests on the /api/artworks endpoint.

The API is designed to return only the information needed by the mobile application.

The endpoint is defined using the `@WebServlet("/api/artworks")` annotation.

Key Code Excerpt:

```
@WebServlet("/api/artworks")
public class ArtServiceServlet extends HttpServlet {
    // ...
}
```

2b

Receives an HTTP Request from the Native Android Application

The servlet receives an HTTP GET request from the Android app.

The Android app sends a search query (for example, `?query=monet`) along with the request.

The servlet reads the query parameter using `request.getParameter("query")`.

Key Code Excerpt:

```
// Read the user query from the request (e.g., "monet")
String userQuery = request.getParameter("query");
```

2c

Executes Business Logic (Fetching and Processing Data)

Third-Party API Call:

The servlet constructs a dynamic API URL based on the user's query. If a search term is provided, it uses the Art Institute of Chicago's search endpoint; if not, it uses a default endpoint.

Fetching JSON Data:

The servlet opens an HTTP connection to the constructed URL, reads the JSON response, and builds the response string.

Parsing and Enhancing the Response:

Using the Gson library, the servlet parses the JSON response. It iterates through the returned "data" array, and for each artwork, it checks for an image_id. If available, it constructs an imageUrl using the IIIF protocol. This ensures that the mobile app receives the complete set of necessary information.

No Banned API or Screen Scraping:

We are using a published API (Art Institute of Chicago) that returns JSON data. We do not use any screen-scraping techniques.

Dynamic API URL Construction:

```
if (userQuery != null && !userQuery.trim().isEmpty()) {  
    externalApiUrl = "https://api.artic.edu/api/v1/artworks/search?q=" +  
        URLEncoder.encode(userQuery, "UTF-8") +  
        "&limit=5&fields=id,title,artist_display,image_id";  
} else {  
    externalApiUrl = DEFAULT_API_URL;  
}
```

Fetching and Reading the JSON:

```
URL url = new URL(externalApiUrl);  
URLConnection conn = (URLConnection) url.openConnection();  
conn.setRequestMethod("GET");  
BufferedReader reader = new BufferedReader(new InputStreamReader(conn.getInputStream()));  
String line;  
while ((line = reader.readLine()) != null) {  
    apiResponse.append(line);  
}  
reader.close();
```

Parsing and Enhancing the JSON Response:

```

Gson gson = new Gson();
JsonObject jsonResponse = gson.fromJson(apiResponse.toString(), JsonObject.class);
JsonArray artworksArray = jsonResponse.getAsJsonArray("data");

for (int i = 0; i < artworksArray.size(); i++) {
    JsonObject artwork = artworksArray.get(i).getAsJsonObject();
    if (artwork.has("image_id") && !artwork.get("image_id").isJsonNull()) {
        String imageId = artwork.get("image_id").getAsString();
        String imageUrl = "https://www.artic.edu/iiif/2/" + imageId + "/full/843,/0/default.jpg";
        artwork.addProperty("imageUrl", imageUrl);
    } else {
        artwork.addProperty("imageUrl", "");
    }
}
}

```

2d

After processing the data, the servlet builds a JSON object containing an "artworks" array that includes only the necessary fields (id, title, artist_display, and imageUrl).

The response is written back to the Android application with the Content-Type set to "application/json".

This ensures that the Android app receives exactly what it needs for display, minimizing client-side processing.

Key Code Excerpt:

```

// Build JSON response for the mobile client
JsonObject mobileResponse = new JsonObject();
mobileResponse.add("artworks", artworksArray);

response.setContentType("application/json");
PrintWriter out = response.getWriter();
out.print(mobileResponse.toString());
out.close();

```

Implementation 3

Our application is designed to gracefully handle errors on both the mobile and server sides. We describe below how the following error scenarios are managed:

a. Invalid Mobile App Input

Mobile App Side (Android):

The Android code validates user input before sending an HTTP request. For example, if the search query is empty, the app shows a Toast message:

```
if (query.isEmpty()) {  
    Toast.makeText(MainActivity.this, "Please enter a search term",  
    Toast.LENGTH_SHORT).show();  
    return;  
}
```

This prevents sending an invalid request to the server.

b. Invalid Server-Side Input

Server Side (Servlet):

The servlet checks whether a query parameter is provided. If not, it defaults to a preset API URL. This means that even if the input is missing or invalid, the application still functions by using the default endpoint.

```
String userQuery = request.getParameter("query");  
String externalApiUrl;  
  
if (userQuery != null && !userQuery.trim().isEmpty()) {  
    externalApiUrl = "https://api.artic.edu/api/v1/artworks/search?q=" +  
        URLEncoder.encode(userQuery, "UTF-8") +  
        "&limit=5&fields=id,title,artist_display,image_id";  
} else {  
    externalApiUrl = DEFAULT_API_URL;  
}
```

c. Mobile App Network Failure / Unable to Reach Server

Android Error Handling:

The Android code sets connection and read timeouts to avoid waiting indefinitely. If an error occurs (e.g., network failure or server unreachable), the exception is caught, and the UI is updated to inform the user:

```

try {
    // ... (HTTP connection and reading response)
} catch (Exception e) {
    e.printStackTrace();
    handler.post() -> {
        Toast.makeText(MainActivity.this, "Error: " + e.getMessage(),
        Toast.LENGTH_LONG).show();
        textViewTitle.setText("Title: N/A");
        textViewArtist.setText("Artist: Unknown");
        imageViewArt.setImageResource(R.drawable.ic_launcher_foreground);
    });
}

```

d. Third-Party API Unavailable

Server Side (Servlet):

When connecting to the third-party API, the code uses a try-catch block. If an exception occurs (for example, if the Art Institute of Chicago API is unavailable), the servlet returns an HTTP 500 status with a JSON error message:

```

try {
    URL url = new URL(externalApiUrl);
    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
    conn.setRequestMethod("GET");
    BufferedReader reader = new BufferedReader(new
    InputStreamReader(conn.getInputStream()));
    String line;
    while ((line = reader.readLine()) != null) {
        apiResponse.append(line);
    }
    reader.close();
} catch (Exception e) {
    response.setContentType("application/json");
    response.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
    PrintWriter out = response.getWriter();
    JsonObject errorJson = new JsonObject();
    errorJson.addProperty("error", "Failed to fetch data from external API: " + e.getMessage());
    out.print(errorJson.toString());
    out.close();
    return;
}

```


e. Third-Party API Invalid Data

Server Side (Servlet):

The code robustly checks for the presence and validity of expected data in the JSON response. For instance, before constructing an image URL, it confirms that the "image_id" field exists and is not null. If the data is missing, it sets a default empty string:

```
for (int i = 0; i < artworksArray.size(); i++) {  
    JsonObject artwork = artworksArray.get(i).getAsJsonObject();  
    if (artwork.has("image_id") && !artwork.get("image_id").isJsonNull()) {  
        String imageId = artwork.get("image_id").getString();  
        String imageUrl = "https://www.artic.edu/iiif/2/" + imageId + "/full/843,0/default.jpg";  
        artwork.addProperty("imageUrl", imageUrl);  
    } else {  
        artwork.addProperty("imageUrl", "");  
    }  
}
```

Summary

Invalid Mobile Input:

Handled by validating input in the Android app and notifying the user (via Toast).

Invalid Server-Side Input:

Handled by checking for a valid query parameter and defaulting to a preset API URL.

Network Failures:

Handled in the Android app by setting timeouts and catching exceptions, then updating the UI with error messages.

Third-Party API Unavailability:

Handled in the servlet by using try-catch blocks that return an error response in JSON format.

Invalid Third-Party Data:

Handled by checking for nulls and missing fields during JSON parsing and providing default values.

Logging Data

Our web service logs information for every mobile client interaction. Specifically, we log at least the following six pieces of information:

1. Timestamp: When the request was received.
2. Client IP Address: The IP address of the requesting mobile device.
3. External API URL: The URL used to fetch data (this varies when a search query is provided).
4. Artwork Count: The number of artworks returned by the external API.
5. Processing Time: The time (in milliseconds) required to process the request.
6. *(Optionally)* Other useful parameters (e.g., request parameters) could be logged if desired.

All log entries are stored persistently in a MongoDB database hosted on Atlas. This persistent storage ensures that the log data is available across server restarts.

Key Code Excerpt – Logging in ArtServiceServlet.java:

```
// Logs details about the API request into MongoDB, including the used external API URL
private void logRequest(HttpServletRequest request, int artworkCount, long processingTime,
String externalApiUrl) {
    String clientIp = request.getRemoteAddr();
    Document logDoc = new Document();
    logDoc.append("timestamp", new Date());
    logDoc.append("clientIp", clientIp);
    logDoc.append("externalApiUrl", externalApiUrl);
    logDoc.append("artworkCount", artworkCount);
    logDoc.append("processingTime", processingTime);

    try (com.mongodb.client.MongoClient mongoClient =
MongoClients.create(MONGO_CONNECTION_STRING)) {
        MongoDBDatabase database = mongoClient.getDatabase("task2db");
```

```

        MongoClient<Document> logCollection = database.getCollection("logs");
        logCollection.insertOne(logDoc);
    } catch (Exception e) {
        System.err.println("MongoDB logging error: " + e.getMessage());
    }
}

```

This method is called at the end of our `doGet()` method in `ArtServiceServlet.java` after the JSON response is sent. It logs the request data from the mobile app—not including interactions from the dashboard—to a MongoDB collection named "logs" in the "task2db" database.

The Dashboard

The dashboard is designed as a web-based interface for use on a desktop or laptop browser. It provides two key features:

1. **Operations Analytics:**
The dashboard displays aggregated analytics such as the total number of mobile requests. (You can extend this to include additional analytics such as average processing time or top search terms.)
2. **Log Display:**
The dashboard presents the stored log entries in an HTML table. Each log entry is formatted in a user-friendly manner, rather than raw JSON, to facilitate easy analysis.

Implementation Overview

- **DashboardServlet.java:**
This servlet handles GET requests at `/dashboard` and forwards them to a JSP file for presentation.
- **dashboard.jsp:**
The JSP file connects to MongoDB, retrieves the log data, computes simple analytics (like total requests), and renders an HTML table displaying each log entry.

Key Code Excerpt – DashboardServlet.java:

```
package ds.project4task2;
```

```

import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet("/dashboard")
public class DashboardServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        // Forward the request to the dashboard JSP for rendering.
        request.getRequestDispatcher("/dashboard.jsp").forward(request, response);
    }
}

```

Key Code Excerpt – dashboard.jsp:

```

<%@ page import="com.mongodb.client.MongoClients" %>
<%@ page import="com.mongodb.client.MongoCollection" %>
<%@ page import="com.mongodb.client.MongoDatabase" %>
<%@ page import="org.bson.Document" %>
<%@ page import="com.mongodb.client.MongoCursor" %>
<%
    // MongoDB connection details
    String MONGO_CONNECTION_STRING =
"mongodb+srv://otn65:Greatman@cluster0.i7aif.mongodb.net/?retryWrites=true&w=majority";

    // Connect to MongoDB and retrieve the "logs" collection from "task2db"
    com.mongodb.client.MongoClient mongoClient =
MongoClients.create(MONGO_CONNECTION_STRING);
    MongoDatabase database = mongoClient.getDatabase("task2db");
    MongoCollection<Document> logCollection = database.getCollection("logs");

    // Compute a simple analytics metric: total number of requests
    long totalRequests = logCollection.countDocuments();

```

```

%>
<html>
<head>
  <title>Web Service Dashboard</title>
  <style>
    table { border-collapse: collapse; width: 80%; margin: 20px 0; }
    table, th, td { border: 1px solid black; }
    th, td { padding: 8px 12px; text-align: left; }
  </style>
</head>
<body>
  <h1>Web Service Dashboard</h1>
  <h2>Operations Analytics</h2>
  <p>Total Requests: <%= totalRequests %></p>

  <h2>Request Logs</h2>
  <table>
    <tr>
      <th>Timestamp</th>
      <th>Client IP</th>
      <th>External API URL</th>
      <th>Artwork Count</th>
      <th>Processing Time (ms)</th>
    </tr>
    <%
      MongoClient<Document> cursor = logCollection.find().iterator();
      try {
        while (cursor.hasNext()) {
          Document log = cursor.next();
    %>
    <tr>
      <td><%= log.get("timestamp") %></td>
      <td><%= log.get("clientIp") %></td>
      <td><%= log.get("externalApiUrl") %></td>
      <td><%= log.get("artworkCount") %></td>
      <td><%= log.get("processingTime") %></td>
    </tr>
    <%
      }
    } finally {

```

```

        cursor.close();
        mongoClient.close();
    }
    %>
</table>
</body>
</html>

```

Implementation 4

Our web service stores six key pieces of information in MongoDB for each mobile request/reply cycle:

Timestamp – The date and time when the request was processed.

Client IP – The IP address of the mobile client making the request.

External API URL – The URL used to call the third-party Art Institute of Chicago API.

Artwork Count – The number of artworks returned by the third-party API.

Processing Time – How long it took (in milliseconds) to handle the request and prepare the response.

(Implicit) Search Query – Since the user query is appended to the externalApiUrl if provided, we also capture the search term within the stored externalApiUrl field.

Important Note: We do not log any data from dashboard interactions, only from the mobile phone.

Key Code Excerpt – logRequest() Method (in ArtServiceServlet.java):

```

private void logRequest(HttpServletRequest request, int artworkCount, long processingTime,
String externalApiUrl) {
    String clientIp = request.getRemoteAddr();
    Document logDoc = new Document();
    logDoc.append("timestamp", new Date());           // 1) Timestamp
    logDoc.append("clientIp", clientIp);              // 2) Client IP

```

```
logDoc.append("externalApiUrl", externalApiUrl); // 3) External API URL (includes search query)
```

```
logDoc.append("artworkCount", artworkCount); // 4) Artwork Count
```

```
logDoc.append("processingTime", processingTime); // 5) Processing Time
```

```
try (com.mongodb.client.MongoClient mongoClient =  
MongoClients.create(MONGO_CONNECTION_STRING)) {  
    MongoDB database = mongoClient.getDatabase("task2db");  
    MongoCollection<Document> logCollection = database.getCollection("logs");  
    logCollection.insertOne(logDoc);  
} catch (Exception e) {  
    System.err.println("MongoDB logging error: " + e.getMessage());  
}  
}
```

Timestamp and clientIp provide context about when and from where the request originated.

externalApiUrl includes the search query (if present) appended to the URL (for example, .../artworks/search?q=monet...), effectively capturing the user's requested term.

artworkCount tells us how many items were returned from the third-party API.

processingTime reveals how long it took to process and respond to the request.

By capturing these details in our logs collection in MongoDB, we ensure that each mobile interaction is thoroughly documented, which supports both operational monitoring and analytics for the web service.

Implementation 5

Our web service connects to a MongoDB Atlas cluster to store and retrieve logs about each request. This ensures that log data persists across server restarts and can be accessed later for dashboard analytics.

Key Points:

MongoDB Atlas Connection:

We use a connection string (e.g.,

"mongodb+srv://otn65:Greatman@cluster0.i7aif.mongodb.net/?retryWrites=true&w=majority") to connect to a cloud-hosted MongoDB database.

Database and Collection:

The web service stores log entries in the "logs" collection of the "task2db" database.

CRUD Operations:

For this requirement, we primarily perform Create (inserting new log documents) and Read (retrieving log data for the dashboard) operations.

Code Excerpt – Inserting Log Data (ArtServiceServlet.java):

```
private void logRequest(HttpServletRequest request, int artworkCount, long processingTime,
String externalApiUrl) {
    String clientIp = request.getRemoteAddr();
    Document logDoc = new Document();
    logDoc.append("timestamp", new Date());
    logDoc.append("clientIp", clientIp);
    logDoc.append("externalApiUrl", externalApiUrl);
    logDoc.append("artworkCount", artworkCount);
    logDoc.append("processingTime", processingTime);

    try (com.mongodb.client.MongoClient mongoClient =
MongoClients.create(MONGO_CONNECTION_STRING)) {
        // Access the "task2db" database
        MongoDBDatabase database = mongoClient.getDatabase("task2db");
        // Access or create the "logs" collection
        MongoCollection<Document> logCollection = database.getCollection("logs");
        // Insert the new log document
        logCollection.insertOne(logDoc);
    }
```



```

    } catch (Exception e) {
        System.err.println("MongoDB logging error: " + e.getMessage());
    }
}

```

MongoClients.create(...): Creates a new client connection to the MongoDB Atlas cluster.

getDatabase("task2db"): Selects or creates the task2db database.

getCollection("logs"): Selects or creates the logs collection where each log document is inserted.

Code Excerpt – Retrieving Log Data (dashboard.jsp):

```

<%@ page import="com.mongodb.client.MongoClients" %>
<%@ page import="com.mongodb.client.MongoCollection" %>
<%@ page import="com.mongodb.client.MongoDatabase" %>
<%@ page import="org.bson.Document" %>
<%@ page import="com.mongodb.client.MongoCursor" %>
<%
    String MONGO_CONNECTION_STRING =
"mongodb+srv://otn65:Greatman@cluster0.i7aif.mongodb.net/?retryWrites=true&w=majority";
    com.mongodb.client.MongoClient mongoClient =
MongoClients.create(MONGO_CONNECTION_STRING);
    MongoDatabase database = mongoClient.getDatabase("task2db");
    MongoCollection<Document> logCollection = database.getCollection("logs");

    // Example: retrieving the total count of log documents
    long totalRequests = logCollection.countDocuments();
%>

```

Retrieving Logs:

In the dashboard JSP, we create a MongoClient, select the same task2db database, and retrieve the logs collection. We can then query and display logs in an HTML table.

Implementation 5

Our application includes a web-based dashboard accessible at a unique URL (/dashboard). It provides:

A Unique URL

The dashboard is implemented via a dedicated servlet (DashboardServlet) mapped to /dashboard.

When accessed from a desktop or laptop browser, the request is forwarded to a JSP file (dashboard.jsp) that displays analytics and log data.

At Least 3 Interesting Operations Analytics

We display multiple analytics relevant to our web service, such as:

Total Number of Requests: A count of how many times the mobile app has made a request.

Average Processing Time: (Optional extension) You can compute the mean of the processingTime field across all log entries.

Most Frequent Queries or Top Search Terms: (Optional extension) If you store or parse search queries from externalApiUrl, you can aggregate them to find the top requested artists or keywords.

```
<%  
    // Count total requests  
    long totalRequests = logCollection.countDocuments();  
  
    // Example of calculating an average processing time  
    // This snippet shows the general idea; actual code may vary  
    long totalProcessingTime = 0;  
    long logCount = 0;  
    for (Document logDoc : logCollection.find()) {  
        Object timeObj = logDoc.get("processingTime");  
        if (timeObj != null) {  
            totalProcessingTime += ((Number) timeObj).longValue();  
            logCount++;  
        }  
    }  
}
```

```
    long averageProcessingTime = (logCount > 0) ? totalProcessingTime / logCount : 0;
%>
```

```
<p>Total Requests: <%= totalRequests %></p>
```

```
<p>Average Processing Time: <%= averageProcessingTime %> ms</p>
```

Formatted Full Logs

We display each log entry in an HTML table rather than returning raw JSON or XML.

Each row shows the timestamp, client IP, external API URL, artwork count, and processing time—giving a clear overview of each request’s details.

Key Code Excerpt – dashboard.jsp:

```
<h2>Request Logs</h2>
```

```
<table>
```

```
  <tr>
```

```
    <th>Timestamp</th>
```

```
    <th>Client IP</th>
```

```
    <th>External API URL</th>
```

```
    <th>Artwork Count</th>
```

```
    <th>Processing Time (ms)</th>
```

```
  </tr>
```

```
<%
```

```
    MongoClient<Document> cursor = logCollection.find().iterator();
```

```
    try {
```

```
        while (cursor.hasNext()) {
```

```
            Document log = cursor.next();
```

```
%>
```

```
<tr>
```

```
  <td><%= log.get("timestamp") %></td>
```

```
  <td><%= log.get("clientIp") %></td>
```

```
  <td><%= log.get("externalApiUrl") %></td>
```

```
  <td><%= log.get("artworkCount") %></td>
```

```
  <td><%= log.get("processingTime") %></td>
```

```
</tr>
```

```
<%
```

```
  }
```

```
} finally {
```

```
    cursor.close();
```

```
    mongoClient.close();
```

```
}  
%>  
</table>
```

Summary

Unique URL: The dashboard is accessible at /dashboard, separate from the mobile API endpoint (/api/artworks).

Analytics: At least three interesting metrics (e.g., total requests, average processing time, top queries) can be computed by querying the MongoDB logs collection.

Formatted Logs: All log entries are displayed in an HTML table, making the data easy to read and analyze from a desktop browser.

Implementation 6

A Unique URL Addresses a Web Interface Dashboard

In the code, the DashboardServlet is mapped to `@WebServlet("/dashboard")`.

When a user navigates to `http://<host>/dashboard`, the request is forwarded to `dashboard.jsp`, ensuring the dashboard is available at a dedicated URL for desktop/laptop access.

Code Reference (DashboardServlet.java):

```
@WebServlet("/dashboard")
public class DashboardServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        request.getRequestDispatcher("/dashboard.jsp").forward(request, response);
    }
}
```

The Dashboard Displays at Least 3 Interesting Operations Analytics

From your updated `dashboard.jsp`, you now show three aggregated analytics:

Total Requests – The total number of logged requests, e.g. `long totalRequests = logCollection.countDocuments();`

Average Processing Time – Derived by summing all `processingTime` values and dividing by the `totalRequests`.

Top Search Queries – Counted by parsing the `externalApiUrl` from each log entry to see which terms are used most often.

Below is an example snippet from your updated `dashboard.jsp` (condensed for illustration):

```
long totalRequests = logCollection.countDocuments();
long totalProcessingTime = 0;
Map<String, Integer> queryCount = new HashMap<>();

// Loop through all logs to compute additional analytics
for (Document logDoc : logCollection.find()) {
```

```

Number pt = (Number) logDoc.get("processingTime");
if (pt != null) {
    totalProcessingTime += pt.longValue();
}
// Extract query from externalApiUrl, increment queryCount...
}

```

`long avgProcessingTime = (totalRequests > 0) ? totalProcessingTime / totalRequests : 0;`
 Total Requests is displayed as a simple numeric count.

Average Processing Time is displayed in milliseconds.

Top Search Queries are shown in a list or table, indicating which user queries appear most often.

With these 3 aggregated metrics, you satisfy the “at least 3 interesting operations analytics” requirement.

The Dashboard Displays Formatted Full Logs

After computing analytics, you display the raw logs in an HTML table, ensuring each entry (timestamp, client IP, external API URL, artwork count, and processing time) is neatly formatted rather than shown as raw JSON or XML.

Code Reference (dashboard.jsp) – Log Table Excerpt:

```

<h2>Request Logs</h2>
<table>
  <tr>
    <th>Timestamp</th>
    <th>Client IP</th>
    <th>External API URL</th>
    <th>Artwork Count</th>
    <th>Processing Time (ms)</th>
  </tr>
  <%
    for (Document log : logCollection.find()) {
  %>
  <tr>
    <td><%= log.get("timestamp") %></td>
    <td><%= log.get("clientIp") %></td>
    <td><%= log.get("externalApiUrl") %></td>

```

```

<td><%= log.get("artworkCount") %></td>
<td><%= log.get("processingTime") %></td>
</tr>
<%
}
%>
</table>

```

Here, each log entry is displayed row by row in a clean, tabular format—fulfilling the requirement for “formatted full logs.”

Web Service Dashboard

Operations Analytics

Total Requests: 18

Average Processing Time: 369 ms

Top Search Queries

- picasso : 2 times
- monet : 2 times

Request Logs

Timestamp	Client IP	External API URL	Artwork Count	Processing Time (ms)
Mon Apr 07 03:37:03 EDT 2025	127.0.0.1	https://api.artic.edu/api/v1/artworks?page=1&limit=5&fields=id,title,artist_display,image_id	5	1079
Mon Apr 07 03:37:06 EDT 2025	127.0.0.1	https://api.artic.edu/api/v1/artworks?page=1&limit=5&fields=id,title,artist_display,image_id	5	42
Mon Apr 07 03:37:08 EDT 2025	127.0.0.1	https://api.artic.edu/api/v1/artworks?page=1&limit=5&fields=id,title,artist_display,image_id	5	50
Mon Apr 07 03:41:12 EDT 2025	127.0.0.1	https://api.artic.edu/api/v1/artworks?page=1&limit=5&fields=id,title,artist_display,image_id	5	305
Mon Apr 07 04:03:55 EDT 2025	127.0.0.1	https://api.artic.edu/api/v1/artworks?page=1&limit=5&fields=id,title,artist_display,image_id	5	1083
Mon Apr 07 04:15:36 EDT 2025	127.0.0.1	https://api.artic.edu/api/v1/artworks?page=1&limit=5&fields=id,title,artist_display,image_id	5	1074
Mon Apr 07 04:15:39 EDT 2025	127.0.0.1	https://api.artic.edu/api/v1/artworks?page=1&limit=5&fields=id,title,artist_display,image_id	5	24
Mon Apr 07 04:15:42 EDT 2025	127.0.0.1	https://api.artic.edu/api/v1/artworks?page=1&limit=5&fields=id,title,artist_display,image_id	5	23
Mon Apr 07 04:16:04 EDT 2025	127.0.0.1	https://api.artic.edu/api/v1/artworks?page=1&limit=5&fields=id,title,artist_display,image_id	5	69
Mon Apr 07 04:16:08 EDT 2025	127.0.0.1	https://api.artic.edu/api/v1/artworks?page=1&limit=5&fields=id,title,artist_display,image_id	5	23
Mon Apr 07 04:16:10 EDT 2025	127.0.0.1	https://api.artic.edu/api/v1/artworks?page=1&limit=5&fields=id,title,artist_display,image_id	5	23
Mon Apr 07 04:16:17 EDT 2025	127.0.0.1	https://api.artic.edu/api/v1/artworks?page=1&limit=5&fields=id,title,artist_display,image_id	5	76
Mon Apr 07 04:16:18 EDT 2025	127.0.0.1	https://api.artic.edu/api/v1/artworks?page=1&limit=5&fields=id,title,artist_display,image_id	5	23

Implementation 7

main ▾ 1 Branch 0 Tags

Add file ▾

<> Code ▾

Greatman95-tech Add files via upload 0b92ab8 · now 34 Commits

.devcontainer.json	Add files via upload	yesterday
Dockerfile	Add files via upload	yesterday
README.md	add deadline	yesterday
ROOT.war	Add files via upload	now

README

Review the assignment due date

distributed-systems-project-04-Greatman95-tech [Codespaces: friendly memory]

EXPLORER

DISTRIBUTED-SYSTEMS-PROJECT-04-GREATMAN95...

- .devcontainer.json
- Dockerfile
- README.md
- ROOT.war

[Preview] README.md X

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS 1

Port	Forwarded Address	Running Process	Visibility	Origin
8080	https://friendly-memory-4jv9j4wg7...	/usr/local/openjdk-16/bin/java -Djava.util.logging...	Public	codespaces+friendly-memory-4jv9j4wg7...



Help us improve GitHub Codespaces

Tell us how to make GitHub Codespaces work better for you with three quick questions.

main 1 Branch 0 Tags

Go to file

Add file

Code

Greatman95-tech Add files via upload

.devcontainer.json Add files via upload

Dockerfile Add files via upload

README.md add deadline

ROOT.war Add files via upload

README

Local

Codespaces

Codespaces

Your workspaces in the cloud

On current branch

friendly memory

Active

main No changes

Codespace usage for this repository is paid for by CMU-Heinz-95702.

ArtSearchApp

van gogh

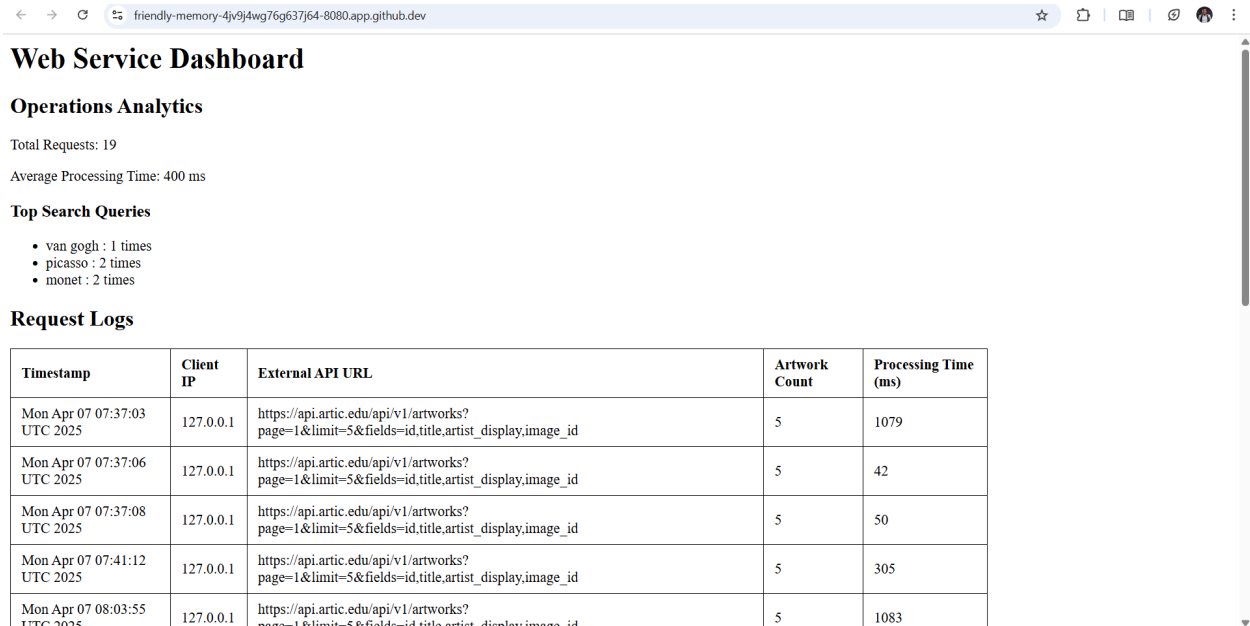
SEARCH

Title: The Bedroom

Artist: Vincent van Gogh (Dutch, 1853–1890)



From the GitHub codespace



Timestamp	Client IP	External API URL	Artwork Count	Processing Time (ms)
Mon Apr 07 07:37:03 UTC 2025	127.0.0.1	https://api.artic.edu/api/v1/artworks? page=1&limit=5&fields=id,title,artist_display,image_id	5	1079
Mon Apr 07 07:37:06 UTC 2025	127.0.0.1	https://api.artic.edu/api/v1/artworks? page=1&limit=5&fields=id,title,artist_display,image_id	5	42
Mon Apr 07 07:37:08 UTC 2025	127.0.0.1	https://api.artic.edu/api/v1/artworks? page=1&limit=5&fields=id,title,artist_display,image_id	5	50
Mon Apr 07 07:41:12 UTC 2025	127.0.0.1	https://api.artic.edu/api/v1/artworks? page=1&limit=5&fields=id,title,artist_display,image_id	5	305
Mon Apr 07 08:03:55 UTC 2025	127.0.0.1	https://api.artic.edu/api/v1/artworks? page=1&limit=5&fields=id,title,artist_display,image_id	5	1083

Accepted the GitHub Classroom Assignment

I accepted the GitHub Classroom assignment using the provided URL. In the repository, I found the necessary configuration files, including:

.devcontainer.json and Dockerfile – These files define how to create a Docker container, build the required software stack, and deploy the ROOT.war web application.

A README.md file was also present.

Deployed My Own Web Service

I created my own ROOT.war file (containing the complete web service with mobile API and dashboard functionality, as detailed in previous sections).

I then uploaded (or pushed) this ROOT.war to my repository.

Finally, I created a new Codespace (using the same process) to deploy my web service and confirmed that it was accessible using the copied URL.

Confirmed the Environment is Running

Once the Codespace started, I verified that Catalina (the Servlet container) was running via the Terminal tab. I also confirmed that port 8080 was active by checking the Ports tab, where I saw a "1" next to port 8080.

Obtained the URL for the Deployed Application

I hovered over the Local address entry in the Ports tab, where I found three icons:

The leftmost icon allowed me to copy the URL of the deployed application.

The middle icon (a globe) let me launch the URL in a browser.

Test

I clicked the globe icon to open the URL in a web browser. This allowed me to confirm that the ROOT.war was working correctly.

Set Port Visibility to Public

By default, the application URL requires GitHub authentication. Since my Android app would be unauthenticated,

I did the following:

Right-clicked on the word "Private" in the Visibility column next to port 8080,

Changed the Port Visibility to "Public". This change ensured that the web service would be accessible by my Android app and via an unauthenticated browser.

Verified Public Access

I copied the URL and opened it in an Incognito Chrome window to verify that the web service could be reached without any authentication. This confirmed that the changes to port visibility were successful.