Project 4 – Photo Wallpaper App

Mitun Adenuga (AndrewID: aadenuga)

Description:

My application takes a keyword input string from the user, and uses it to fetch and display a picture from the Unsplash API.

Here is how my application meets the task requirements

1. Implement a native Android application

The name of my native Android application project in Android Studio is: WallpaperAppAndroid

1. Has at least three different kinds of views in its Layout ( )

My application uses TextView, EditText, Button, and ImageView. Take a look at the content\_main.xml and activity\_main.xml for details of how they are incorporated into the LinearLayout.

Here is a screenshot of the layout before the picture is fetched.

A screen shot of a cell phone

Description automatically generated

Here is a screenshot of the layout after the picture is fetched.

A screen shot of a cell phone

Description automatically generated

Here is a screenshot of the wallpaper after selecting set as wallpaper

A screen shot of a cell phone

Description automatically generatedA screen shot of a cell phone

Description automatically generated  
c. Makes an HTTP request (using an appropriate HTTP method) to your web service

My Android app makes an HTTP GET request to my web service using the HttpURLConnection class. This is implemented in the GetWallpaper class, specifically in the search method. The app constructs a URL with the search query and sends a GET request to your servlet.

URL url = **new** URL(SERVLET\_URL + "?query=" + encodedQuery);

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

connection.setRequestMethod("GET");

connection.connect();  
  
Where the SERVLET\_URL is defined as <http://10.0.2.2:8080/AndroidPicture-1.0-SNAPSHOT/getWallpaper> within the Android application.

With the deployment to Codespaces that URL is now defined as <https://redesigned-memory-xqgw5j7pjvqc9p6v-8080.app.github.dev>.

In the browser (from the servlet) an example URL would be <http://localhost:8080/AndroidPicture-1.0-SNAPSHOT/getWallpaper?query=nature>

d. Receives and parses an XML or JSON formatted reply from your web service:

After making the HTTP request, the app receives a JSON response from the web service. It then uses the JSON Object class to parse this response. The app looks for an "imageUrl" field in the JSON, which contains the URL of the wallpaper image.

JSONObject jsonResponse = new JSONObject(response.toString());

if (jsonResponse.has("imageUrl")) {

String imageUrl = jsonResponse.getString("imageUrl");

return BitmapFactory.decodeStream(new URL(imageUrl).openConnection().getInputStream());

}  
  
An example of the response is   
**{**"imageUrl": "<https://images.unsplash.com/photo-1465146344425-f00d5f5c8f07?crop=entropy&cs=tinysrgb&fit=max&fm=jpg&ixid=M3w2NzA4ODl8MHwxfHNlYXJjaHwxfHxuYXR1cmV8ZW58MHx8fHwxNzMyMTI3NzkxfDA&ixlib=rb-4.0.3&q=80&w=1080>"**}**

e. Displays new information to the user

Once the JSON is parsed and the image URL is extracted, the app downloads the image and displays it to the user. This is done in the MainActivity class, specifically in the wallpaperReady method. The app updates an ImageView with the new wallpaper image, making it visible to the user. The app also sets the homepage to the new picture.

public void wallpaperReady(Bitmap picture) {

ImageView pictureView = (ImageView) findViewById(R.id.wallpaperImage);

if (picture != null) {

pictureView.setImageBitmap(picture);

pictureView.setVisibility(View.VISIBLE);

}

// ... other UI updates

}  
  
This method updates the UI with the new wallpaper.

f. Is repeatable (I.e. the user can repeatedly reuse the application without restarting it.)

The app allows users to perform multiple searches without restarting. This is achieved through the click listener set on the submit button in MainActivity. Each time the user enters a new search term and clicks the button, a new search is initiated, allowing for repeated use of the application.

1. Implement Web Service
2. Implement a simple (can be a single path) API.

The WallpaperPicture servlet implements a simple API with two endpoints: "/getWallpaper" and "/dashboard". The main functionality is handled by the "/getWallpaper" endpoint, which processes search queries for wallpapers.  
  
b. Receives an HTTP request from the native Android application

The servlet is set up to handle GET requests. When the Android app makes a request to the "/getWallpaper" endpoint with a query parameter, the servlet processes this request in the handleGetWallpaper method.

c. Executes business logic appropriate to your application. This includes fetching

XML or JSON information from some 3rd party API and processing the response.

The servlet I implemented executes the following business logic:

* It receives a search query from the Android app.
* It then makes a request to the Unsplash API (a 3rd party API) to fetch images based on the search query:

URL url = **new** URL(UNSPLASH\_API\_URL + "?query=" + encodedQuery + "&client\_id=" + ACCESS\_KEY);

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

* The servlet processes the JSON response from Unsplash:

JSONObject jsonResponse = **new** JSONObject(content.toString());

JSONArray results = jsonResponse.getJSONArray("results");

* It extracts the URL of the first image from the results:

String imageUrl = firstImage.getJSONObject("urls").getString("regular");

* The servlet then sends this image URL back to the Android app in a JSON format:

JSONObject responseJson = **new** JSONObject();

responseJson.put("imageUrl", imageUrl);

out.print(responseJson.toString());

* Additionally, the servlet logs each request to a MongoDB database, including details like timestamp, client IP, user agent, and the search query.

**Web Service Logging and Analysis Dashboard Requirements**

**4. Log useful information**

The WallpaperServlet logs various useful information for each request. This includes:

* Timestamp of the request
* Client IP address
* User Agent (browser/device information)
* Search query
* Status of the request (SUCCESS, BAD\_REQUEST, NO\_RESULTS, NETWORK\_ERROR, JSON\_ERROR, ERROR)
* Processing time of the request
* Image URL (for successful requests)
* Error messages (for failed requests)

**5. Store the log information in a database**

* A MongoDB connection is established in the init() method using a connection string.
* The database and collection are initialized: database = mongoClient.getDatabase("myFirstDatabase") and logsCollection = database.getCollection("request\_logs").
* For each request, a log entry is created as a MongoDB Document and inserted into the collection: logsCollection.insertOne(logEntry).

**6. Display operations analytics and full logs on a web-based dashboard**

a. A unique URL addresses a web interface dashboard for the web service.

The dashboard is accessible via the "/dashboard" endpoint of your web service.

b. The dashboard displays at least 3 interesting operations analytics.  
The code fetches and calculates the following analytics:

* Total number of requests: long totalRequests = logsCollection.countDocuments();
* Number of successful requests: long successfulRequests = logsCollection.countDocuments(new Document("status", "SUCCESS"));
* Recent logs (last 10 requests): List<Document> recentLogs = logsCollection.find().sort(new Document("timestamp", -1)).limit(10).into(new ArrayList<>());

c. The dashboard displays **formatted** full logs.

The recent logs fetched from the database are passed to a JSP page for display:  
request.setAttribute("recentLogs", recentLogs);The dashboard data is then forwarded to a JSP page for rendering:  
request.getRequestDispatcher("/dashboard.jsp").forward(request, response);While the actual JSP file is not provided in the search results, the code suggests that it would render the analytics and logs in a formatted manner.