Criterion C : Product Development and Techniques

Techniques Employed:

1. Recycler View
2. JSON Parsing
3. Coroutines
4. Google Maps API/ Places API
5. Setting up a Database: Appwrite

---

1. RecyclerView: PlacesItemRecylerViewAdapter Java Class

The RecyclerView is a powerful and widely used Android component that allows developers to efficiently display large collections of items in a list or grid format. In the context of the TravelBuddy app, the RecyclerView is used to display a list of saved places to explore.
One of the main benefits of using the RecyclerView is its flexibility and performance. It is designed to efficiently handle large data sets and can recycle views to improve performance. It also supports animations and user interactions such as scrolling and swiping.

The RecyclerView uses an adapter to manage the data and a layout manager to define how the items are displayed on the screen.

In the TravelBuddy app, the PlacesItemRecyclerViewAdapter class is used as the adapter for the RecyclerView. This adapter takes a list of Places objects as input and creates a ViewHolder for each item in the list. The ViewHolder is responsible for holding a reference to each view in the item layout, so that the views can be efficiently recycled as the user scrolls through the list.

Here is a code snippet from the PlacesItemRecyclerViewAdapter class that demonstrates how the adapter is created and bound to the RecyclerView:

```java
public class PlacesItemRecyclerViewAdapter extends
RecyclerView.Adapter<PlacesItemRecyclerViewAdapter.ViewHolder> {

   private final List<Places> mValues;

   public PlacesItemRecyclerViewAdapter(List<Places> items) {
       mValues = items;
   }

   @NonNull
   @Override
   public ViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int
viewType) {
       FragmentItemBinding binding =
FragmentItemBinding.inflate(LayoutInflater.from(parent.getContext()), parent,
false);
       return new ViewHolder(binding);
   }

   @Override
   public void onBindViewHolder(final ViewHolder holder, int position) {
       Places places = mValues.get(position);
       holder.mItem = places;
       holder.mIdView.setText(places.getName());
       holder.mContentView.setText(places.getAddress());
   }

   @Override
   public int getItemCount() {
       return mValues.size();
   }

   public class ViewHolder extends RecyclerView.ViewHolder {
       public final TextView mIdView;
       public final TextView mContentView;
       public Places mItem;

       public ViewHolder(FragmentItemBinding binding) {
           super(binding.getRoot());
           mIdView = binding.name;
           mContentView = binding.address;
       }

       @Override
       public String toString() {
           return super.toString() + " '" + mContentView.getText() + "'";
       }
   }
}
```

In this code, the onCreateViewHolder method is called when the RecyclerView needs a new ViewHolder to display an item. The method inflates the layout for the item using the FragmentItemBinding class and returns a new ViewHolder instance.

The onBindViewHolder method is called when the RecyclerView needs to bind data to a ViewHolder. This method retrieves the Places object for the current position and uses it to populate the views in the ViewHolder. Finally, the getItemCount method returns the total number of items in the list.

2. JSON Parsing: Account Fragment Kotlin Class

JSON (JavaScript Object Notation) is a lightweight data format used to exchange data between a client and a server. It's easy to read and write, and it's supported by many programming languages..

JSON parsing is the process of converting JSON data (JavaScript Object Notation) into usable objects in your code. In the code provided, JSON parsing is used to extract the data from the response object received from the Appwrite API and convert it into usable objects in the form of Places objects.

```
   // Convert the response object to a JSON object
   val jsonResponse = JSONObject(response.toJson())
   val documents = jsonResponse.getJSONArray("documents")

   for (i in 0 until documents.length()) {
       val document = documents.getJSONObject(i)

       // Extract the fields from the data object
       val data = document.getJSONObject("data")
       val name = data.getString("name")
       val address = data.getString("address")
       val latitude = data.getDouble("latitude")
       val longitude = data.getDouble("longitude")
       // Create a Places object and add it to the list
       val place = Places(name, address, latitude, longitude)
       places.add(place)
       // Convert the place object to a String

       // Show a Toast message with the placeString
       Log.d("Appwrite fetch place", place.toString())

   }
} catch (e: AppwriteException) {
   e.printStackTrace()
}
```

In the code snippet , response is the object received from the Appwrite API, which is then converted to a JSONObject using the toJson() method provided by the Appwrite SDK. The JSON object is then used to extract an array of documents using the getJSONArray() method. For each document in the array, the code extracts the fields for name, address, latitude, and longitude from the "data" object using getXXX() methods provided by the JSONObject class. The extracted fields are then used to create a Places object, which is added to a list of Places objects. The PlacesItemRecyclerViewAdapter then uses this list to populate the RecyclerView with data.

3. Coroutines: Account Fragment [Kotlin Class]

Coroutines are a concurrency design pattern that allows developers to write asynchronous code in a synchronous style. Coroutines can be used to perform long-running tasks such as network requests or database queries without blocking the main thread. The time complexity of using Coroutines depends on the number of tasks performed and the processing power of the device. However, because Coroutines are designed to be lightweight and efficient, they generally have a low impact on performance.

This feature of Kotlin enables asynchronous programming, making it easier to perform long-running tasks in a non-blocking manner. They allow developers to write asynchronous code in a sequential and readable manner, making it easier to reason about and debug. In the provided code, coroutines are used to perform network operations, such as fetching data from the Appwrite database, without blocking the main UI thread.

In the onCreateView method, a coroutine is launched to fetch the places from the Appwrite database, and once the data is retrieved, it is passed to the PlacesItemRecyclerViewAdapter for display in the RecyclerView.

In the onResume method, another coroutine is launched to fetch the places from the Appwrite database. This method is called when the fragment is resumed, such as when the user navigates back to the AccountFragment from another fragment.

In the fetchPlacesFromAppwrite method, a coroutine is used to make an asynchronous request to the Appwrite database to retrieve the places saved by the user. The result of the request is parsed into a list of Places objects, which is returned by the method. The use of coroutines in this method allows the network operation to be performed in a non-blocking manner, ensuring that the app remains responsive while the data is being retrieved.

```
override fun onCreateView(
       inflater: LayoutInflater, container: ViewGroup?,
       savedInstanceState: Bundle?
): View? {

   val view = inflater.inflate(R.layout.fragment_item_list, container, false)

   val logoutButton = view.findViewById<Button>(R.id.logout_button)
   logoutButton.setOnClickListener {
       lifecycleScope.launch {
           logoutAndNavigateToMainActivity()
       }
```

```
    }

    // Find the RecyclerView by its ID
    recyclerView = view.findViewById<RecyclerView>(R.id.list)
    val context = recyclerView.context
    recyclerView.adapter = null
    recyclerView.recycledViewPool.clear()

    recyclerView.layoutManager = GridLayoutManager(context, mColumnCount)

    // Launch a coroutine to fetch places
    lifecycleScope.launch {
        val places = fetchPlacesFromAppwrite()
        recyclerView.adapter = PlacesItemRecyclerViewAdapter(places)
    }

    return view
}
override fun onResume() {
    super.onResume()
    lifecycleScope.launch {
        val places = fetchPlacesFromAppwrite()
        recyclerView.adapter = PlacesItemRecyclerViewAdapter(places)
    }
}
```

In the above code snippets, I used lifecycleScope.launch to launch coroutines that fetch data from the Appwrite database and update the RecyclerView adapter with the retrieved data. By launching these coroutines, I was able to perform these time-consuming tasks on a background thread without blocking the UI thread. This helps to ensure that the app remains responsive and performs smoothly even when dealing with large amounts of data.

---

4. Google Maps API/ Places API: ThingstoDoCategoryAdapter Class

To use Google Maps API/Places API, I had to obtain an API key from the Google Cloud Console. So, I went to the Google Cloud Console (https://console.cloud.google.com/) and set up an API key.

```
package com.example.travelbuddy;
import com.example.travelbuddy.AppwriteUserHelper;
```

```
public class MapRecommendationActivity extends FragmentActivity implements
OnMapReadyCallback {
   private static final String PLACES_API_BASE_URL =
"https://maps.googleapis.com/maps/api/place/nearbysearch/json";
```

The code uses the Google Maps API and Places API to display nearby places based on a selected category, and allows users to add those places to a list that is stored in an Appwrite database. Here are some of the key techniques used in the code:

1. Initializing the Google Maps API and Places API: In the onCreate method, the SupportMapFragment is obtained and getMapAsync is called to initialize the GoogleMap object. The PlacesClient is also initialized with the RectangularBounds and TypeFilter options.
2. Requesting location permissions: The requestLocationPermissions method is called to request permission from the user to access their location. This is necessary to show the user's current location on the map and search for nearby places based on their location.
3. Obtaining the user's current location: Once location permission is granted, the getCurrentLocation method is called to get the user's current location using the FusedLocationProviderClient. A marker is added to the map to show the user's current location, and the getNearbyPlacesBasedOnCategory method is called to search for nearby places based on the selected category.
4. Fetching nearby places based on category: The getNearbyPlacesBasedOnCategory method uses the Nearby Search API to search for nearby places based on the selected category within a specified radius. The API call is made using the OkHttpClient and Request objects, and the response is parsed using a JSONObject and JSONArray. A marker is added to the map for each nearby place.
5. Adding places to the Appwrite database: The addPlaceToAppwrite method is called when the user clicks the "Add to list" button on a marker's info window. The place name, address, latitude, longitude, and user ID are stored in an HashMap object and added to the Appwrite database using the databases.createDocument method.

---

5. <u>Setting up a Database: Appwrite</u>

Appwrite is a backend as a service platform that provides a range of services such as authentication, database, storage, and serverless functions. The database service is used to store and manage data for the application. Appwrite provides a simple API for accessing the database, and it supports multiple database types such as MongoDB and MySQL. The time complexity of using the Appwrite database depends on the size of the data stored, the number of requests made, and the processing power of the server.

```
   Map<String, Object> placeData = new HashMap<>();
   placeData.put("name", name);
   placeData.put("address", address);
   placeData.put("latitude", latitude);
   placeData.put("longitude", longitude);
placeData.put("userId", userId);


   String uniqueDocumentId = UUID.randomUUID().toString();

   databases.createDocument(
           "642481c8a9ed2d76e6ef",
           "642481d1ba3a9bd5359b",

           uniqueDocumentId,
           placeData,
           new CoroutineCallback<>((result, error) -> {
               if (error != null) {
                   Log.d("Appwrite", "error" + name + address + latitude +
longitude);

                   error.printStackTrace();
                   return;
               }

               Log.d("Appwrite", result.toString());
               Log.d("Appwrite", "Place added successfully:");
           })
   );

}
```

This code block is part of the MapRecommendationActivity class which creates a new document in an Appwrite database with the details of a place that was clicked on the map. The code creates a new HashMap named placeData to store the name, address, latitude, longitude, and userId of the place.
Then, a unique document ID is generated using UUID.randomUUID().toString(). The createDocument() method is called on the Databases object from the Appwrite client to create a new document in a specific collection. The collection ID and document ID are passed as parameters, along with the placeData HashMap.

Finally, a callback function is passed to the createDocument() method to handle the result or any errors that may occur during the document creation process. If there is an error, it is logged and the function returns. Otherwise, a success message is logged to the console.

References and Sources (Citations with Necessary Links)

Google Maps API
Google Maps Platform Documentation: https://developers.google.com/maps
Getting Started Guide: https://developers.google.com/maps/gmp-get-started

Appwrite for Database
Appwrite Official Website: https://appwrite.io/
Getting Started with App Write: https://appwrite.io/docs

Android Developers Guide: Introduction and implementation guide for RecyclerView.
https://developer.android.com/develop/ui/views/layout/recyclerview

Android Developers Documentation: Documentation on RecyclerView, including version history and features. https://developer.android.com/jetpack/androidx/releases/recyclerview

Jetpack Android Developers: Updates on RecyclerView, including significant changes like MergeAdapter renaming. https://developer.android.com/jetpack/androidx/releases/recyclerview

JSON Parsing
Mozilla MDN Web Docs on JSON: JSON Guide.
https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON

Kotlin Coroutines
Kotlinlang.org on Coroutines: Kotlin Coroutines Guide.
https://kotlinlang.org/docs/coroutines-guide.html

Google Places API
Google Places API Documentation: Google Places API Guide.
https://developers.google.com/maps/documentation/places/web-service/overview