Boldizsar Banfia MAD Discussion

COM527 Mobile Application Development

1. How you have used XML layout files in your application

XML layout files are used in Android app development to define the layout and UI components for an app's activities or views. An XML layout file specifies the structure and appearance of the UI components, such as buttons, text views, and images, as well as how they are arranged on the screen.

Within on create in each activity, the xml file with the layout is initialized  .

In the application, there are several XML layout files being used.

Each activity has its XML files that defines the layout, in which Linear layout was used throughout. Depending on the orientation selected, Android's LinearLayout layout manager places its child views in a single column or row. In accordance with the order, they are defined in the XML layout file, child views are arranged one after the other, either horizontally or vertically. Consistently in the application the “match\_parent” parameter defined the even space, height, width in each layout.

In the main\_activity.xml the MapView component is set to have a width and height of "match\_parent". This means that the MapView will stretch to fill the entire width and height of its parent layout, which is the LinearLayout in this case.

The addpoi\_layout.xml defines a form for the user to input information about a place. It includes three "EditText" that are similar in function to an input tag in HTML. Each EditText and Button component is set to have a width of "match\_parent", ergo they expand horizontally to fill the entire width of the parent LinearLayout. The height of each EditText and Button component is set to "wrap\_content", which means the height will adjust to fit the content within the view. The LinearLayout itself has a height of "match\_parent", which means it will stretch to fill the entire height of its parent view. Therefore, the height of the EditText and Button components will be determined by the height of their content and the padding and margins applied to them, but they will all expand vertically to fill the available height of the parent LinearLayout. The “android:hint” acts as a placeholder until there is an input, much like in HTML. XML in this case serves the purpose of both HTML and CSS in a web application. Another example of usage is when the marker popup is displayed, in which case the text property is defined in the program and replaced by it, instead of extracting the value of it. These Id resources are contained within the ids.xml. Contrary to that, strings.xml contains immutable string resources for the UI and massaging of the app, like menu.xml that contains the menu option.

These XML layout files are used by the Android framework to inflate the UI components when the corresponding activity or view is displayed. The framework reads the XML layout file and generates the necessary objects and properties to create the UI components on the screen. The code can then manipulate these components as needed to interact with the user or display information.

1. Accessing user interface elements with their ID from your Kotlin code and handing events

General way of accessing an UI element is via id. As the syntax demonstrates  it works identically to ‘getElementById’ in JavaScript, the ‘finViewById’ locates the in the file, it requires two elements as an argument, one generic type ‘<EditText>’ and ‘R.id.editText1’ the unique id assigned to it. After that  the input value was extracted, converted to the appropriate format, and stored. Another way, that an interface element accessed was the same way  with the slight difference. The attribute in this case is being programmatically changed.

Also accessing another UI element works within xml file for example  when the menu layout accesses another xml file ‘strings.xml’ where the value of the element is declared. This approach makes the application modular. Once there is a reference to the view, the properties can be manipulated, listen for user events, and perform other actions. This allows to create dynamic and interactive user interfaces in an Android app.

Lambda function was used as event listener 

It is not only concise but it allows to capture variables from surrounding scope and proved efficient SAM conversions. Akin to a click listener method in JavaScript, the wrapped code is executed when the button is pressed. At some cases event listener is completely handled by a function such us the menu inflater  . As extra to the assessment, another event listener was added to the map. It is using the osmoroid exclusive class that includes the two different event listener, one single tap that detects whether a marker was pressed or a long press that takes the latitude and longitude of the clicked place and initiates the point of interest or POI creating process.

1. Communication between activities using Intents and Bundles, and request codes where appropriate.

As instructed, the extraction of POI attributes was implemented on a separate activity. To initiate the launch of the second activity Intent was used.

 . An Intent is an object in Android that is used to request an action from another app component, such as launching an activity or sending a broadcast. It can carry data and specify the target component. The intent takes two variables, the current class (this) and the destination ‘AddPoi’ activity. Since the app is not pure Kotlin the .java class could not be left out. It reads the result back  and only then will the block of code be executed. If the user presses the back button, the result will not be sent back. In ‘AddPoi’ activity, after the user input was stored, to send data back the putExtra() method was used. his method adds data to the Intent as extras using a key-value pair. The keys used to store the data in the Intent extras are the same keys used to retrieve the data in the receiving activit. When ‘AddPoi’ activity finishes, it sets the result of the activity using the setResult() method with a result code of RESULT\_OK and a new Intent that contains the data to be returned as extras. When the activity result is returned, the lambda function checks if the result code is RESULT\_OK. If so, it retrieves the returned Intent using result.data. Then the attributes are retrieved from the intent extras using the corresponding key, assigned to variables, and added to the list.

1. Use of preferences

The activity initiates from the menu, with the already mentioned method, creating a new Intent object that specifies the Preferences activity as the target. The startActivity() method is then called to launch the preferences screen, and true is returned to indicate that the menu item click event has been handled. The preference file was defined in the preference.xml file similarly to a layout file. Simple checkbox preference with a Boolean value. It is necessary to construct a new activity to display these settings in a real app. A PreferenceFragment, which is a portion of the Preferences Activity that will house the preference selections, must be included in this or more specifically, a PreferenceFragmentCompat for backwards-compatibility reasons. In the onCreate() method, it gets a reference to the FragmentManager by calling supportFragmentManager, and then it starts a transaction to replace the existing content of the activity with a new instance of the MyPrefsFragment class. This fragment is a subclass of PreferenceFragmentCompat, which is a library class that provides a framework for building preferences screens. In the MyPrefsFragment class, we override the onCreatePreferences() method, which is called when the preferences screen is being created. In this method, the addPreferencesFromResource() method used to load the preferences from an XML resource file.

Lastly, the preference Boolean value is retrieved  in onResume when the focus comes back to the main activity. The variable will be further utilized in a condition of the post request.

1. Persisting and retrieving data using an on-device SQLite database

The Room application with one database table contains three separate classes plus the main activity.

Data entity class representing the actual data.  the database table that will be used to store the objects that this class represents is specified. The data class has a constructor that takes all of six properties as arguments, and the properties are declared using shorthand syntax in the class declaration.

The DAO as the @Dao annotation indicates that, used for accessing and manipulating data in a local database. The interface defines two methods, getAllPoi() The method specifies a SQL query, returns a list of PoiEntity objects, which represents all of the points of interest in the database. Secondly, insert, this method is annotated with @Insert, which tells Room to generate the necessary code for inserting a new PoiEntity object into the poi table. The method takes a single PoiEntity object as its argument and returns a long value, which represents the primary key of the newly inserted row. The general role of the DAO (The Data Access Object) is an interface or abstract class that defines the methods for accessing and manipulating data in the local database. The DAO acts as an intermediary between the business logic of an application and the underlying database. The DAO provides a higher-level abstraction layer over the low-level SQLite database, making it easier to perform CRUD (Create, Read, Update, and Delete) operations on the data. The DAO interface also provides a clean separation between the database logic and the application logic, making it easier to test and maintain the codebase.

Database object class is responsible for defining the database schema, creating, and managing the database connections, and exposing DAO instances for accessing the data. The class provides a getDatabase() method that returns a single instance of the database using the Singleton pattern. This method creates a new database instance or retrieves an existing one using the Room.databaseBuilder().

It all initialized first in the main activity within the menu. First it creates a database instance, loops through the attributes,  launches a coroutine, then  in the background adds it to the database.

1. Network communication, including both GET and POST requests as appropriate.

The HTTP request is made easier with the help of the Fuel library. It offers a simple and clear syntax for processing requests and answers. When the corresponding menu item is clicked, the code first sets the URL to fetch data from the URL. The application uses the Fuel HTTP client library to send a GET request to the URL and handle the response. The "result" parameter  is a Result object that contains either the success data or the error data returned by the web service. The "fold" method  of the Result object is used to handle both the success and failure cases. The response is handled using a callback function that takes three arguments: the HTTP request, the HTTP response, and the result data. The response data is obtained as a byte array and converted to a string. The string is then parsed as a JSON array and each JSON object in the array is processed to extract the properties. These properties are used to create a new "PoiEntity" object, which is added to the "poiList" collection.  This is iterating through the list placing a marker onto the map with the details of each object.

The POST request takes place in the onResume(), since it depends on the preferences, whether the it is being utilised or not. After the checking for null values of the global variables, the code creates a new JSONObject and adds the POI information to it by creating list pair objects that represents the POST data. Each item of POST data is a Pair object containing a key/value pair. Then, the Fuel library is used to make a POST request to a web service endpoint with the URL. The body of the request is set to the string representation of the JSON object, and the "Content-Type" header is set to "application/json". The response of the request is handled using a lambda expression passed to the response() method. The response can be either a success or an error, and the result.fold() method is used to handle both cases. If the request is successful, a Toast message is displayed with the response message. Otherwise, an error message is logged to the console and displayed as a Toast message.

1. JSON parsing

To parse JSON, Android has a selection of classes. By doing this, Java objects will be created from JSON that has been loaded into memory. Two classifications are employed. JSON Object that represents an individual JSON object, that has been created at post request A screenshot of a computer

Description automatically generated with low confidence very well exemplifies it. In this case the use of JSON parsing allows the client application to send the POI information to the web service in a structured and standardized format that is easy to parse on the server side.

The other one is the JSON Array represents an array of JSON objects. It has been utilized at the get request. In this case, the response returned by the REST API is expected to be in JSON format. The response data is first converted to a string using  the String(data) function. This is necessary because the response data is received in bytes, which need to be converted to a string before they can be parsed as JSON. Then it is parsed as a JSON array using the  JSONArray(responseJson) constructor. The resulting JSONArray object contains a list of JSON objects, each representing a point of interest (POI). To extract the required data from each JSON object, the code uses the getJSONObject(i) method to get each object in the array, and then uses the getString()  and getDouble() methods to extract the required fields from the object. These fields are then used to create a PoiEntity object, which represents a POI. By using JSON parsing, the program is able to extract data from the REST API response in a structured manner.

1. Use of fragments

As instructed, the two fragments are simultaneously visible, achieved by directly include fragments within the layout, by embedding the  . Both fragments are contained in the main activity xml following Model-View-ViewModel (MVVM) architecture pattern. The two fragments acted as the views, while the two view models acted an intermediate between the view and the actual models, which are the DAOs. Data sharing between fragments are done solely using view models and live data. There is two view model, the ‘TestViewModel’ with custom setters that was used to store the current location of the user, which was intercepted in the main by the GPS listener and uploaded  . It was then retrieved in frag1 

and stored. Frag1 captures user input (name, type, description) and validates it. Upon clicking the submit button, the setOnClickListener retrieves the input data. After the data turned into the right format  , fragment gets the database, creates a poiEntity instance and insert a row in the background with these data and resets the text boxes. Note that null data A picture containing text, font, screenshot

Description automatically generated can not be uploaded since the program will not upload if the variable is empty.

The second, nn Frag2, the poiViewModel instance is obtained using the viewModels() Kotlin property delegate. This ViewModel retrieves all points of interest from the database and exposes them as a LiveData<List<PoiEntity>>. In Frag2, an Observer instance for poiViewModel.getAllPoi() observes the list of POIs. When a new POI is added to the database, the list is updated, and the Observer is triggered. For each POI in the updated list, Frag2 instantiates a new Marker object with the corresponding latitude and longitude from the POI entity and adds it to the overlays of the map, using map1.overlays.add(marker).

The updateFinalLat() function in Frag2 centers the map at the user's current location by setting the map's center using map1.controller.setCenter(GeoPoint(finalLat, finalLon)), where finalLat and finalLon are updated by the livelat and livelon Observer instances.

In essence, Frag1 sends data to the database through the ViewModel, while Frag2 observes the data from the ViewModel and updates the map accordingly. This architecture facilitates communication and data sharing between both fragments without them having to directly interact with each other. Similarly, to Ajax, it displays everything from the database without having to refresh the page.