Android Development with Kotlin

Networking

JSON API

JSON stands for JavaScript Object Notation and is a light-weight data exchange format that is used to exchange data over a network. An API stands for Application Programming Interface, and consist of a set of protocols and rules that are used to create an application.

JSON APIs are APIs that are used to transmit data over a network using a JSON format. These APIs follow a particular structure for their objects and this structure is the model over which the application is based.

JSON API delineates how the clients should request/edit data from a server.

A large number of JSON APIs are available online for use, such as the movie db api. Some of these APIs require an API authentication key that is used to track the API usage and regulate and prevent malicious use of the API's data. This key can also be used to uniquely identify an application and track the API usage for an application.

API Access

The internet is filled with numerous sites and a number of them offer their services via APIs as well. But not all of these APIs are open access i.e. they require an authentication key that can be generated on their respective websites. These authentication keys can be used to monitor the use of the API by an application to regulate malicious use as well as to keep track of the data used, which is often used for billing purposes.

These keys can also be used to implement various access levels for an API. For example, different developers can have access to different features of the APIs based on the plan they register for. The developer who created the API may have full access where as a developer with a free usage plan maybe allowed to access only a few features based on the rules of the API.

Fetching Data from an API

To use an API in your Android Application, you should follow the following steps:

(The following example uses themoviedb.org API)

1. Register for the API Key online.
2. Save your API key in the gradle.properties with the name TMDB_API_KEY. Then add the following code to your module build.gradle file.

```
android {
    buildTypes.each {
        it.buildConfigField 'String', 'TMDB_API_KEY', TMDB_API_KEY
    }
}
```

3. Create a basic app structure such that the Application has a Room database corresponding to your API and required ViewModels and fragments for your project.
4. Add retrofit libraries to the build.gradle for your project module.

```
//Retrofit
implementation 'com.squareup.retrofit2:retrofit:2.7.0'
implementation 'com.squareup.retrofit2:converter-gson:2.7.0'
```

5. Create an API Service interface with suspend functions for the GET, POST or PUT requests and a companion object with a lazy instance for the interface. The instance is created using the following:
   a. An interceptor that attaches the "api_key" to every request.
   b. A http client built using the interceptor.
   c. A gson builder that sets the formats for specific data types. (optional)
   d. A Retrofit builder that uses the httpClient and gson builder to create the instance.

   The following code creates an instance of the TmdbService within its companion object:

```
private val retrofitService by lazy {
    // Add api key to every request
    val interceptor = Interceptor { chain ->
        val request = chain.request()
        val url = request.url().newBuilder()
            .addQueryParameter(
                "api_key",
                BuildConfig.TMDB_API_KEY
            )
            .build()
        val newRequest = request.newBuilder()
            .url(url)
            .build()
        chain.proceed(newRequest)
    }

    //creates the http client
    val httpClient =
         OkHttpClient().newBuilder().addInterceptor(interceptor).b
         uild()

    //gson builder sets date format
    val gson = GsonBuilder()
        .setDateFormat("yyyy-MM-dd HH:mm:ss")
        .create()

    //creates the instance of the TmdbService class
    Retrofit.Builder()
        .baseUrl(BASE_URL)
        .client(httpClient)
        .addConverterFactory(GsonConverterFactory.create(gson))
        .build()
        .create(TmdbService::class.java)
}
```

   The GET function to fetch the latest movies can be written as

```
@GET("discover/movie?certification_country=US&adult=false&" +
"vote_count.gte=100&with_original_language=en&sort_by=primary_rele
ase_date.desc")
suspend fun getMovies(): Response<TmdbMovieList>
```

The parameter for the @GET is the url for the api request. Similar patterns re followed for @POST and @PUT request where the data is also attached to the request.

The functions of the retrofit interface must return objects of type Response, and the actual result can be later extracted from the Response object.

To explore more about the different types of Retrofit requests, please visit: https://square.github.io/retrofit/

6. Add @SerializedName("**name**") to your data classes such that the serialized name matches the key for the data object within your JSON API.
7. The data from the retrofit service should be used to update the Room database. And the data to be displayed within the fragment should be fetched from the room database. This practice avoids a complex fetching architecture for data requests. This can be done in the repository as follows:

```
suspend fun fetchFromNetwork(){
    val result = tmdbService.getMovies()
    if(result.isSuccessful){
        val movieList = result.body()
        movieList?.let{ movieListDao.insertMovies(it.results) }
    }
}
```

8. **To display images**

   a. Add glide to your project

```
//Image loading from network
implementation 'com.github.bumptech.glide:glide:4.10.0'
annotationProcessor 'com.github.bumptech.glide:compiler:4.10.0'
```

   b. use the following code to fetch images from an image URL

```
Glide.with(containerView)
    .load(TmdbService.POSTER_BASE_URL + movie.posterPath)
    .error(R.drawable.poster_placeholder)
    .into(movie_poster)
```

the R.darwable.poster_placeholder image is loaded if a network fetching error occurs, otherwise, the resulting image is loaded into the movie_poster image view.

Status and Error handling

Network fetching errors may occur if the internet connection is unavailable or is significantly slower. This can lead to the crashing of your application if these messages aren't handled properly. The following code segment can be used to handle such errors and display appropriate status messages to ensure that the user is notified of the application processes.

Loading Status Class

```
enum class Status {
    LOADING,
    SUCCESS,
    ERROR
}
enum class ErrorCode {
    NO_DATA,
    NETWORK_ERROR,
    UNKNOWN_ERROR
}

data class LoadingStatus(val status: Status, val errorCode: ErrorCode?, val
message: String?) {
    companion object {
        fun loading(): LoadingStatus {
            return LoadingStatus(
                Status.LOADING,
                null,
                null
            )
        }
        fun success(errorCode: ErrorCode? = null, msg: String? = null)
                                                  : LoadingStatus {
            return LoadingStatus(
                Status.SUCCESS,
                errorCode,
                msg
            )
        }
        fun error(errorCode: ErrorCode, msg: String? = null): LoadingStatus
{
            return LoadingStatus(
                Status.ERROR,
                errorCode,
                msg
            )
        }
    }
}
```

The fetchFromNetwork function for the Repository should be modified such that it returns an instance of the LoadingStatus class. This instance is then observed by the calling class and is handled according to display the related status message within the activity/fragment.

fetchFromNetwork() function

```
suspend fun fetchFromNetwork() =
    try {
        val result = tmdbService.getMovies()
        if (result.isSuccessful) {
            val movieList = result.body()
            movieList?.let { movieListDao.insertMovies(it.results) }
            LoadingStatus.success()
        } else{
            LoadingStatus.error(
                ErrorCode.NO_DATA)
        }
    } catch(ex: UnknownHostException){
        LoadingStatus.error(
            ErrorCode.NETWORK_ERROR)
    } catch(ex: Exception){
        LoadingStatus.error(
            ErrorCode.UNKNOWN_ERROR, ex.message)
    }
```

Swipe and Refresh

To refresh the data on a swipe, wrap your recycler view within a SwipeRefreshLayout and add an onRefreshListener to the layout which deletes the previously saved data and fetches new data from the network.

This can be done as follows:

Layout File

```
<androidx.swiperefreshlayout.widget.SwipeRefreshLayout
    android:id="@+id/swipe_refresh"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/movie_list"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".ui.TaskListFragment"
        tools:listitem="@layout/list_item"
        app:layoutManager="androidx.recyclerview.widget.GridLayoutManager"
        app:spanCount="2"
        />
</androidx.swiperefreshlayout.widget.SwipeRefreshLayout>
```

onRefreshListener

```
swipe_refresh.setOnRefreshListener{
    viewModel.refreshData()
}
```

To study more about the Swipe and Refresh Layout visit:
https://developer.android.com/training/swipe/add-swipe-interface