09/09/23

Android basics

- Activity lifecycle: 'onCreate()' method is called when we enter a new Activity I.e., our 'MainActivity'. Other methods ('onPause()', 'onDestroy(),' 'onSaveInstanceState(),' 'onRestoreInstanceState()' etc.) are also called when other things happen to the app - screen rotations, navigating away from the app or moving to another screen

Building layouts in Android – XML vs Compose

- **XML:** Designing the UI with XML mark-up and then connecting it to code
- **Compose:** A newer library used to create UI's, there is no mark-up involved. It is done alongside Kotlin code using @Composable functions

Compose basics

- Where to put Compose code
  - Directly into the 'setContent{}' method in 'onCreate()' method
  - In a @Composable function that can be used somewhere else by calling it (in the Preview or in 'setContent{}' in the 'onCreate()' method
- Composable functions
  - Small blocks of UI code that we can re-use in other areas by calling the function
  - Being able to pass arguments to Composable functions to use/display a value that has been passed in - I.e., display a name String passed into a Composable function
- Preview function
  - This function allows us to see our UI changes quickly instead of having to run in an emulator, it is usually at the bottom of the file with the '@Preview' annotation. To run a Preview, click the button in the margin beside the Preview.
- Containers
  - Components
    - **Column:** Items within are stacked on top of each other (one below the other)
    - **Row:** items within are stacked side by side
    - **Box:** items within are stacked over each other (on top of each other – can block views below)
  - Alignment
    - Columns and Rows: use the 'verticalAlignment'/'horizontalArrangement' or 'horizontalAlignment'/'verticalArrangement' attributes, use Modifier.fillMaxSize() for the modifier attribute
    - Box: use the 'contentAlignment' attribute
- Compose components
  - **Text:** Displaying text with 'Text()'
  - **Lists:** Displaying a List with 'LazyColumn()'
    - Creating lists in Kotlin using 'listOf()'
  - **Buttons:** Displaying a button using 'Button()'
    - Printing something to the Logcat using the 'Log.d()' method

- **Images:**
  - Displaying an image using 'Image()'
    - PNGs dragged into the 'res/drawable' folder and referenced in code
    - Vector assets by right clicking the drawable folder and creating one from an SVG or using the Clip Art library
    - From a URL using the Coil library for Android
  - Using different modifiers and attributes on an Image
    - Clipping into a shape using '.clip()' modifier
    - Using the 'contentScale' attribute to adjust the Image scaling

Tips

- Importing dependencies
  - Follow the same format for other dependencies in the 'build.gradle.kts' (Module:app) file for 'implementation' dependencies.
  - Dependencies are often provided on libraries that can be copied in directly
- Filtering through the Logcat
  - Using the correct level based on the Log method used (debug for Log.d())
  - Filter on the Tag
- Adding permissions in the Manifest
  - Internet permission to load Images from a URL if using the Coil library

Reading/Resources:

- Compose theory: https://developer.android.com/jetpack/compose/mental-model#:~:text=Using%20Compose%2C%20you%20can%20build,which%20displays%20a%20greeting%20message.
- Jetpack Compose Playground: https://foso.github.io/Jetpack-Compose-Playground/

Android Notes: 16/09/23

Compose Navigation

- To navigate between screens, we need a NavHost and a NavController. The NavHost is a single place where all our screens are held. The NavController allows us to navigate to any of these.
- To create a NavHost, we use the 'NavHost()' Composable function with arguments for the 'navController', 'startDestination' and 'builder'. The 'navController' must be passed into the NavHost so that it knows about all screens in the app. The 'startDestination' specifies which screen we will start on, and the builder is where we put all our Composable screens.
- To create a NavController, we use the 'rememberNavController()' function and put it in a variable so it can be passed into other functions.
- With each screen added, we must specify its unique 'route' and pass in the navController so that we're able to navigate away.
- To navigate when we're on a screen, we can use 'navController.navigate(<route>)'

Note: To use Compose navigation we must import the dependency in our build.gradle (app) file.

Showing and hiding views on the screen

- Use a 'MutableState' Trigger with a Boolean value in a Composable function
- When the user performs an action I.e., clicks a view, we flip the value of the trigger, the MutableState value will change and the Composable will be recomposed (I.e., it's code will be executed again).
- We can surround the view with an if statement to check the value of the trigger and display either a filled in Like button or an outlined Like button

Kotlin basics

- Variables
    - 'var' and 'val' – use var for hold values that can be changed and val for values that don't change
    - You can specify the type of the variable after a Colon I.e., 'var aString: String = "random string"'
    - Some types that can be stored variables: Int, String, Double, Float, Boolean etc.
- Conditionals
    - If-else statements
    - When statements – these allow us to filter through the different values a variable may have and do something
- Functions
    - Functions help us break code up to make it reusable. Each function should have a specific purpose/responsibility.
    - Naming should be camel case I.e. 'exampleMethod()'

    - Functions can take in parameters and return values. An example: ‘fun add(value1: Int, value2: Int) : Int’
- Loops
    - For loops: ‘for <item> in <collection> {}’ to iterate over a list
    - While loops: use a variable to set a condition on the while loop to continue execution, then adjust the variable on each iteration until it fails the condition I.e., a count variable (so that the loop stops)

Reading/Resources

- Compose Navigation: https://developer.android.com/jetpack/compose/navigation
- Kotlin basics: https://kotlinlang.org/docs/basic-syntax.html

Making Network requests

- To make network requests we can use the Retrofit library for Android
- To set up the request, we must create a Retrofit client, Service and Model
    - **Retrofit client:** will have the Base URL and the GSON converter factory which decodes the JSON coming back from the API into models that we create in our project.
    - **Service:** is an interface that contains method declarations that are annotated with the endpoints of the API I.e., for 'https://hp-api.onrender.com/api/characters,' we use '@GET("characters")'
    - **Model:** is a data class that holds all the fields that we want to deserialize from JSON into.
        - "@SerializedName(<key>)" annotation – the <key> value in this must match key in the JSON that is returned
        - Return types for the fields must correspond to the type in JSON: List<> for arrays, String for text, etc.
- Network requests must be done on a background thread. We can use Kotlin Coroutines to execute tasks on a background thread. To do this:
    - Add 'suspend' before the method in our Service
    - Call the method using a CoroutineScope (this will execute the task on a background thread)

App architecture

- Separating our app into layers can help with keeping our code organised and testable.
- We can introduce a ViewModel layer that is between the UI and the data sources (I.e., network)
    - A ViewModel has a lifecycle just like Android Activities do, and they often outlive activities so can hold onto data after an activity is destroyed/reconfigured
    - They are also useful for sharing data between screens if both screens use the same ViewModel
- A ViewModel often has an observable variable/reference at the top that the UI can access.
    - We can use 'MutableState' here that works with Compose, when the value inside the MutableState changes, the Compose views that use it will be recomposed and updated
- The ViewModel is where we can have logic to make network requests using our Retrofit client. When we are in the ViewModel, we can use a 'viewModelScope.launch{}' to launch a Coroutine that is tied to the lifecycle of the ViewModel. This is so the Coroutine is cleared when the ViewModel is destroyed.
- To connect our ViewModel to our Composable screens, we can initialize it in our MainActivity and pass down the state held in it to each screen so that the screens are able to use the data.

Passing values between screens

- To pass values between screens we can adjust our composable routes (in our NavHost builder) to take in an argument. This can be of any type but is usually a String (so that a unique ID can be passed across, which the receiving screen can use to either make a query to get the full data set with, or filter the value held in a variable in a shared ViewModel to get the required piece of data)
- When we are scrolling through a list, we can pass in the id of the item that has been clicked. This can allow us to display more detailed information in the other screen if we filter the data held in the ViewModel (that both screens use).

Reading/Resources

- ViewModel: https://developer.android.com/topic/libraries/architecture/viewmodel
- Passing data between screens: https://developer.android.com/jetpack/compose/navigation#nav-with-args
- Retrofit: https://www.howtodoandroid.com/retrofit-android-example-kotlin/

Android Notes: 30/09/23

Error handling

- Error responses may come back from the API when making a network request I.e., '404: Not found'
- We can catch these errors by using a try/catch block, and we can narrow down the error by checking the error code of the response in the Exception, and then showing something to user based on the response. We can also log the error using 'Log.e()'
- To show something to the user when we catch an error, we can change the State variable that holds data for the screens. We can add another variable to it for displaying an error and change the value from false to true. The State is observed by the UI and will run through a conditional block of code to determine what should be shown to the user.
- We can also add a 'loading' variable to the State to display a loading progress indicator in between network requests.

Testing

- We can test the different layers of app I.e., Network layer, ViewModel, UI. To test each layer, we need to be able to pass everything that class/layer needs when we create the class. I.e., if our ViewModel needs a HttpClient to make network requests, it must be passed in from somewhere else. This is so that we can create instances of these classes in our tests and pass in 'mocks' for objects like a HttpClient where we can specify what responses we want when we call a method.
- **API testing**
    - To test the API, we will need to use a 'MockWebServer' to act as our server when we make network requests.
    - We can specify the body and response code that we expect back when we make a call to our 'Service.' We can set the body to a JSON object/file that is in our project.
    - To verify that our API works correctly, we compare what we get back from our MockWebServer matches the body/response we set it to respond with when we make a call using our Service.
- **ViewModel testing**
    - To test our ViewModel we will need to mock our Service using the Mockito library and pass it into a new ViewModel instance that we create in our test.
    - We can then 'stub' the response of the service when we call the 'getCharacters()' method to return a list of characters that we specify in the test.
    - When we make 'getCharacters()' call on the ViewModel we can verify that we get the list of characters that we expected.
    - We will need to use 'runTest{}' to execute our tests on a background thread, this is provided by the 'kotlinx-coroutines-test' library.
    - We will also need to switch our Coroutine dispatcher in tests when executing methods that run in a 'viewModelScope.'

- **UI testing**
    - To test our UI, we can use a ‘ComposeTestRule’ which will act as a container that will hold our Composable and allow us to perform user actions on it.
    - We can pass in our own data (that the screen will display) in the ‘setContent{}’ method
    - In our test, we can use the ‘ComposeTestRule’ to scroll to various positions in the list and verify what exists at those positions matches what we have passed in.

Reading/Resources

- Testing fundamentals: https://developer.android.com/training/testing/fundamentals
- Testing Coroutines: https://developer.android.com/kotlin/coroutines/test