Questions on Project:

Wall-D

**what is dto package**

The DTO package stands for Data Transfer Object package. It is a design pattern used to transfer data between different layers of an application, especially between the presentation layer and the service layer or the service layer and the persistence layer. DTOs are simple objects that contain data without any business logic.

**what is SerializedName**

@SerializedName is an annotation used in Java, commonly with the Gson library, to specify the JSON key name that should be mapped to a particular Java field when converting between JSON and Java objects. It helps in cases where the JSON key names do not match the Java field names or when you want to use a different name in your Java code.

**What is Entity Annotation in Room**

Annotations Used in Room:

@Entity:

Marks a class as an entity. Each entity represents a table within the database.

@PrimaryKey:

Identifies the primary key in an entity. The autoGenerate attribute can be used to auto-increment the ID.

@Dao:

Marks an interface or abstract class as a Data Access Object (DAO). DAOs contain methods for accessing the database.

@Insert, @Update, @Delete:

Annotation methods in DAOs to insert, update, or delete entities.

@Query:

Specifies a SQL query. Can be used for read and write operations.

@Database:

Marks an abstract class as a Room database and lists the entities and version number.

**what is mapper package**

In the MVVM (Model-View-ViewModel) architecture, the mapper package is used to handle the conversion of data between different layers of the application. This package typically contains classes and interfaces responsible for transforming data models to and from different formats. This ensures a clear separation of concerns and improves code maintainability and readability.

Purpose of the Mapper Package

Data Transformation: Converts data models from one layer to another. For example, mapping a network response model (DTO) to a domain model used within the application logic.

what is extension function

what is companion object

**what is const val vs val**

const val

Compile-time constant: const val is used to define compile-time constants. The value must be known at compile time and cannot be changed.

Top-level or object declarations: const val can only be used at the top level of a file or within an object declaration (companion object included).

Primitive types and String: const val can only be used with primitive types (Int, Long, Float, etc.) and String.

Example:

kotlin

Copy code

const val MAX\_VALUE = 100

const val BASE\_URL = "https://api.example.com"

Val

Runtime constant: val is used to define read-only properties that are assigned a value at runtime. Once assigned, the value cannot be changed, but it can be assigned through a more complex expression.

Any scope: val can be used at any scope (local variables, class properties, etc.).

Any type: val can be used with any type, not just primitives and strings.

**What is @GET and @POST in retrofit**

**what is suspend function**

A suspend function in Kotlin is a function that can be paused and resumed at a later time without blocking the current thread. It is a key feature provided by Kotlin's coroutines, which are used for asynchronous programming.

Suspend functions can be cancelled by cancelling the coroutine in which they are invoked. They can also check for cancellation using coroutineContext.isActive or by using withContext and ensureActive functions.

**what is @Query in retrofit**

In Retrofit, @Query is an annotation used to define query parameters for HTTP requests. It allows you to specify query parameters that are appended to the URL of a GET request.

**what is dependency injection**

1. \*\*@HiltAndroidApp\*\*: Annotates the Application class to generate the required Dagger-Hilt components. It initializes the Dagger-Hilt dependency injection framework in the application.

2. \*\*@InstallIn\*\*: Specifies where a particular dependency should be available (e.g., Application, Activity, Fragment, Service). It is applied to Dagger-Hilt modules or component classes.

3. \*\*@Module\*\*: Annotates a class to indicate that it provides dependencies. Methods within a module annotated with `@Provides` provide instances of objects to be injected.

4. \*\*@Provides\*\*: Annotates methods within a module to define how to create or provide instances of objects. These methods must return the type of the object being provided.

5. \*\*@Singleton\*\*: Indicates that a dependency should have a singleton scope, meaning that only one instance of the object will be created and shared throughout the application.

6. \*\*@Binds\*\*: Annotates abstract methods in a module to declare which implementation to use when providing an interface or abstract class.

7. \*\*@Component\*\*: Annotates an interface to specify the connections between modules and Android classes. Dagger-Hilt generates an implementation of this interface to perform dependency injection.

8. \*\*@EntryPoint\*\*: Annotates an interface or an object to create an entry point for dependencies that are not supported by Hilt.

9. \*\*@EntryPointAccessors\*\*: Generates accessors for entry points to allow getting dependencies from entry points.

10. \*\*@ViewModelInject\*\*: Annotates the ViewModel constructor to indicate that Dagger-Hilt should provide dependencies to the ViewModel through constructor injection.

11. \*\*@AssistedInject\*\*: Used to annotate the constructors of classes whose instances are created by Dagger-Hilt but require constructor parameters that are not available to Dagger. Works in conjunction with `@Assisted` for ViewModel constructors.

12. \*\*@Inject\*\*: Annotates fields, constructors, or methods to indicate that they should be injected with dependencies by Dagger-Hilt. Used for field injection, constructor injection, or method injection.

13. \*\*@Assisted\*\*: Used in conjunction with `@ViewModelInject` or `@AssistedInject` for classes that require constructor parameters. It marks constructor parameters that will be passed manually when creating the instance.

**what is domain layer**

In the MVVM (Model-View-ViewModel) architecture pattern for Android applications, the domain layer represents the core business logic and functionality of the application. It serves as an abstraction layer that encapsulates the business rules, data manipulation, and operations specific to the domain of the application.

**what is repository in MVVM**

In the MVVM (Model-View-ViewModel) architecture pattern, a repository serves as an intermediary between the ViewModel and the data sources, such as a local database, network service, or any other data provider. It abstracts the logic of retrieving and persisting data, providing a clean API for ViewModel to interact with the data sources without directly coupling it to specific implementation details.

**what is repository layer**

In the MVVM (Model-View-ViewModel) architecture pattern, the repository layer serves as an intermediary between the ViewModel and the data sources, such as a local database, network service, or any other data provider. It encapsulates the logic for handling data operations, including fetching, storing, updating, and deleting data, while abstracting away the details of data retrieval and persistence from the ViewModel.

what is glideImage

Glide is a popular open-source image loading and caching library for Android applications. It is designed to efficiently load and display images from various sources, such as network URLs, local storage, resources, or content providers, while providing features like caching, resizing, transformations, and animated GIF support.

-ContentScale.Crop

Scaling and Cropping: ContentScale.Crop scales the image to fill the layout's dimensions while preserving its aspect ratio. It then crops any excess parts of the image that do not fit within the layout.

**Jetpack Compose Navigation**

BottomBar Navigation

-rememberNavcontroller

-rememberSaveable

-mutableStateOf

-navBackStackEntry - currentBackStackEntryAsState()

currentDestination.hierarchy

popUpTo(navController.graph.findStartDestination().id)

NavHost

navController

NavGraphBuilder

Composable

Jetpack Compose Navigation simplifies navigation in Android apps. It's built on top of the Navigation component from the Android Jetpack library but tailored for Compose.

1. \*\*NavHost\*\*: It's a container that hosts composables associated with navigation destinations. It defines the navigation graph and manages the swapping of screens.

2. \*\*NavController\*\*: It orchestrates navigation within a NavHost. It handles actions like navigating to a destination, popping the back stack, or accessing the current destination.

3. \*\*NavGraph\*\*: It's a collection of navigation destinations organized in a hierarchy. It defines possible paths users can take within an app.

4. \*\*NavDestination\*\*: Each screen or destination in the navigation graph is represented by a NavDestination. It contains information like the destination's ID, arguments, and navigation actions.

5. \*\*NavBackStackEntry\*\*: It represents an entry in the navigation back stack. It holds information about a destination and allows you to retrieve its arguments or lifecycle state.

Setting up Jetpack Compose Navigation involves creating a NavHost, defining a NavGraph with NavDestinations, and using the NavController to navigate between destinations. Additionally, you can use actions, arguments, and deep links for more advanced navigation scenarios.

**observeAsState()**

In Jetpack Compose, observeAsState() is a function used to observe a LiveData or StateFlow object from a ViewModel and automatically recompose the UI whenever the observed data changes.

rememberModalBottomSheetState()

**Remember**

Retaining State: The remember function allows you to create and retain state within a composable function. This state persists across recompositions of the composable, ensuring that the stateful information is preserved even as the UI is updated.

Immutable State: The state created with remember is immutable, meaning that it cannot be directly modified. Instead, you typically use state update functions like mutableStateOf or derivedStateOf to update the state value.

**rememberCoroutineScope()**

rememberCoroutineScope() is a function provided by Jetpack Compose that allows you to create and remember a coroutine scope within a composable function. This coroutine scope is tied to the lifecycle of the composable and ensures that coroutines launched within it are automatically cancelled when the composable is removed from the composition or recomposed.

**modifier in compose**

In Jetpack Compose, modifiers are used to apply various transformations and styling to composables. They allow you to modify the appearance, layout, and behavior of UI elements.

lamda function

**ViewModel**

-viewModelScope.launch(Dispatchers.IO + coroutineExceptionHandler)

viewModelScope: This is a CoroutineScope provided by the ViewModel class in Android Jetpack. It's typically used for launching coroutines tied to the lifecycle of the ViewModel. Coroutines launched within this scope are automatically cancelled when the ViewModel is cleared, helping to avoid memory leaks.

launch: This is a coroutine builder used to start a new coroutine. It launches a new coroutine and returns a reference to the coroutine's Job, which can be used to control the coroutine or to cancel it.

(Dispatchers.IO + coroutineExceptionHandler): This part specifies the context in which the coroutine runs and any exception handling strategy.

Dispatchers.IO: This specifies that the coroutine should run in the IO dispatcher. This dispatcher is optimized for I/O-bound tasks, such as network or disk operations. It's suitable for blocking or long-running operations that do not involve CPU-intensive work.

coroutineExceptionHandler: This is a custom exception handler that handles exceptions thrown by the coroutine. It's typically used to define how to handle exceptions that occur during the execution of the coroutine. For example, it might log the exception or perform error handling logic

**sharedViewModel**

sharedViewModel is a concept in Android Jetpack's ViewModel architecture where multiple fragments within the same activity can access and share data through a common ViewModel instance. This allows fragments to communicate and share data without directly referencing or coupling with each other. The shared ViewModel is typically scoped to the parent activity's lifecycle, ensuring that it persists across configuration changes and that all associated fragments receive consistent data updates. This pattern promotes better separation of concerns and facilitates the development of modular, reusable components within an Android app.

DownloadManager

-setMimeType

NewsApp

postValue

The postValue() method in MutableLiveData is used to update the stored value of the MutableLiveData from a background thread. This is important because LiveData is designed to be used in conjunction with the UI, and UI updates should only be performed on the main (UI) thread to ensure thread safety and prevent potential race conditions.