Gestione dati in Android

ViewModel

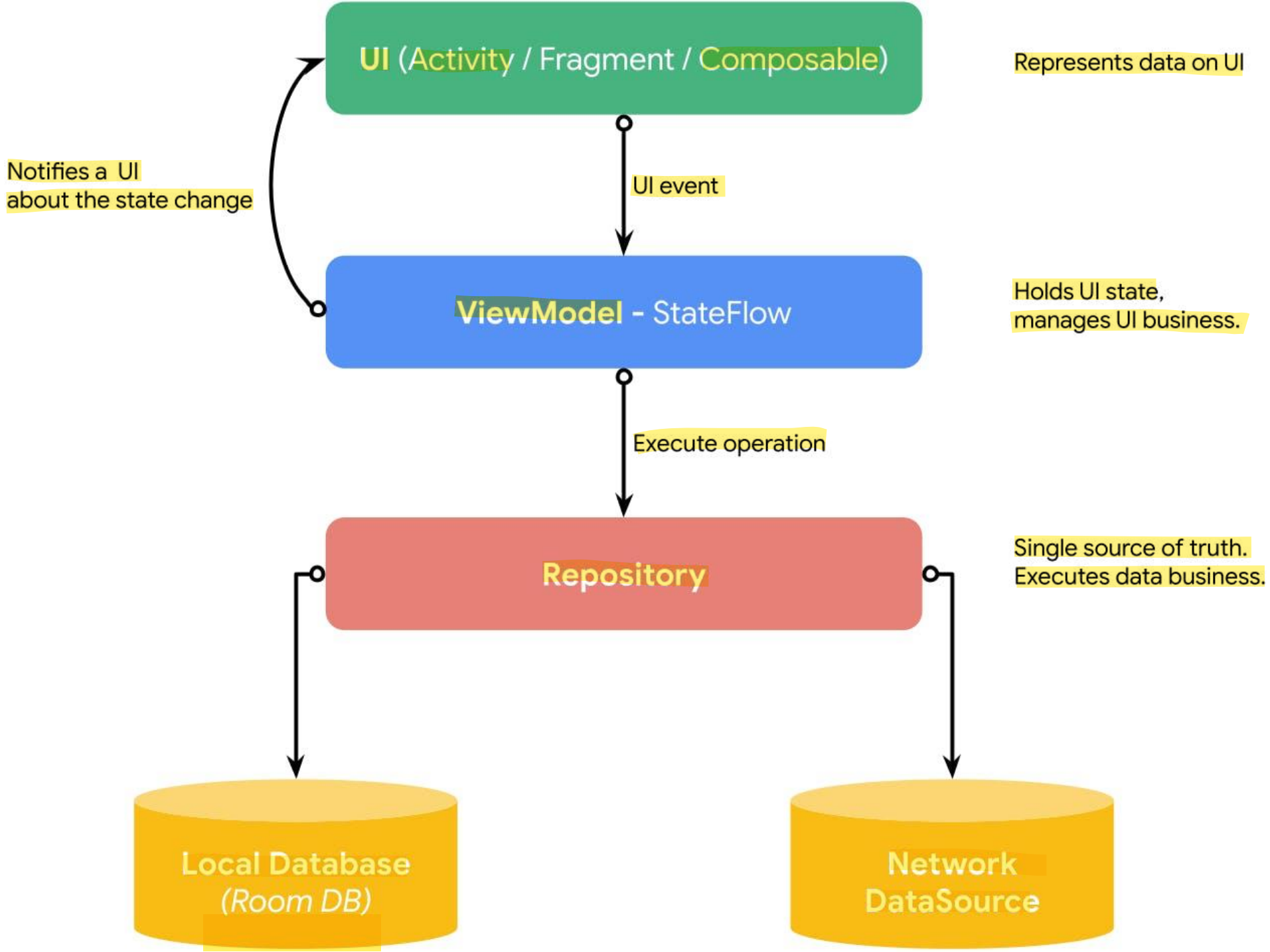
Lezione 8.2

App data e files: Overview

- Android utilizza un file system simile ai file system basati su disco di altre piattaforme
- Il sistema offre diverse opzioni per salvare i dati dell'app:
  1. App-specific storage
     - archivia i file destinati esclusivamente all'uso dell'app, in directory dedicate all'interno di un volume di archiviazione interno o in diverse directory dedicate all'interno dei un volume di archiviazione esterna. NB: Utilizzare le directory all'interno della memoria interna per salvare informazioni riservate a cui altre app non dovrebbero accedere
  2. Shared storage:
     - archivia i file che la tua app intende condividere con altre app, inclusi file multimediali, documenti e altri file
  3. Preferences:
     - archivia i dati primitivi, privati in coppie chiave-valore (key-value data)
  4. Database:
     - archivia i dati strutturati in un database privato utilizzando la libreria di persistenza Room

App data e files: Overview

- Android utilizza un file system simile ai file system basati su disco di altre piattaforme
- Il sistema offre diverse opzioni per salvare i dati dell'app:
  1. App-specific storage
     - archivia i file destinati esclusivamente all'uso dell'app, in directory dedicate all'interno di un volume di archiviazione interno o in diverse directory dedicate all'interno dei un volume di archiviazione esterna. NB: Utilizzare le directory all'interno della memoria interna per salvare informazioni riservate a cui altre app non dovrebbero accedere
  2. Shared storage:
     - archivia i file che la tua app intende condividere con altre app, inclusi file multimediali, documenti e altri file
  3. Preferences:
     - archivia i dati primitivi, privati in coppie chiave-valore (key-value data)
  4. **Database:**
     - **archivia i dati strutturati in un database privato utilizzando la libreria di persistenza Room**


UI (Activity / Fragment / Composable)
Represents data on UI
Notifies a UI
about the state change
UI event
ViewModel - StateFlow
Holds UI state,
manages UI business.
Execute operation
Repository
Single source of truth.
Executes data business.
Local Database
(Room DB)
Network
DataSource

ViewModel

- La classe ViewModel è business logic or screen level state holder
  - Espone lo stato all'interfaccia utente e incapsula la logica di business correlata
- Il suo vantaggio principale è che memorizza nella cache lo stato e lo «persiste» attraverso le modifiche alla configurazione
  - Ciò significa che l'interfaccia utente non deve recuperare nuovamente i dati durante la navigazione tra le attività o dopo le modifiche alla configurazione (es. rotazione dello schermo)

Approfondimento!

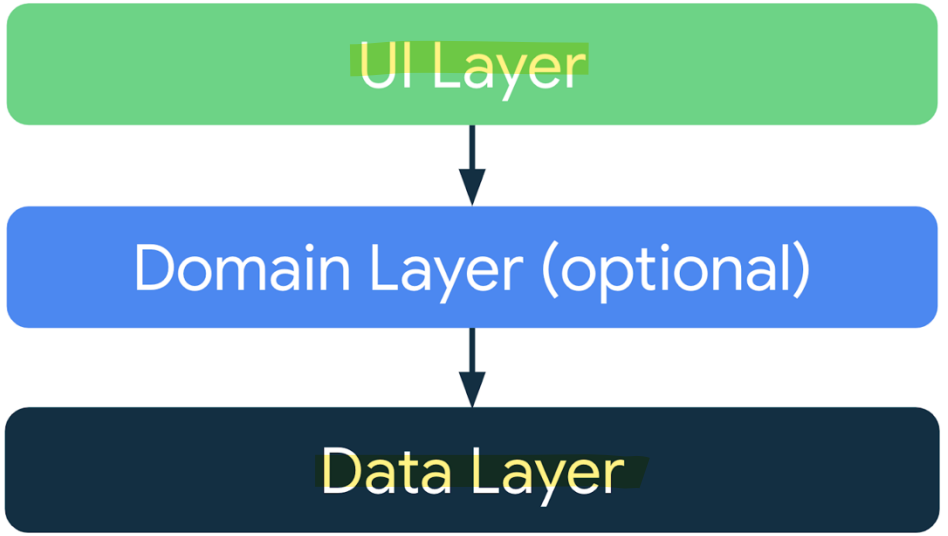
- L'architettura di un'app fornisce linee guida per aiutarti ad allocare le responsabilità dell'app tra le classi
- Un'architettura dell'app ben progettata ti aiuta a scalare la tua app e ad estenderla con funzionalità aggiuntive
- L'architettura può anche semplificare la collaborazione in team
- I principi architetturali più comuni sono:
  - **separation of concerns**
  - **driving UI from a model**

Approfondimento!

- Il principio di progettazione **separation of concerns** afferma che l'app è suddivisa in classi di funzioni, ciascuna con responsabilità separate
- Il principio **driving UI from a model** afferma che dovresti «driving» l'interfaccia utente da un modello, preferibilmente un modello persistente
  - I modelli sono componenti responsabili della gestione dei dati per un'app. Sono indipendenti dagli elementi dell'interfaccia utente e dai componenti della tua app, quindi non sono influenzati dal ciclo di vita dell'app e dai problemi associati

Approfondimento!

- Architettura dell'app consigliata:
  - Considerando i principi architetturali comuni menzionati nella sezione precedente, ogni app dovrebbe avere almeno due livelli:
    - Livello UI: un livello che visualizza i dati dell'app sullo schermo ma è indipendente dai dati
    - Livello dati: un livello che archivia, recupera ed espone i dati dell'app.
      - Puoi aggiungere un altro livello, chiamato livello di dominio, per semplificare e riutilizzare le interazioni tra l'interfaccia utente e i livelli di dati. Questo livello è facoltativo

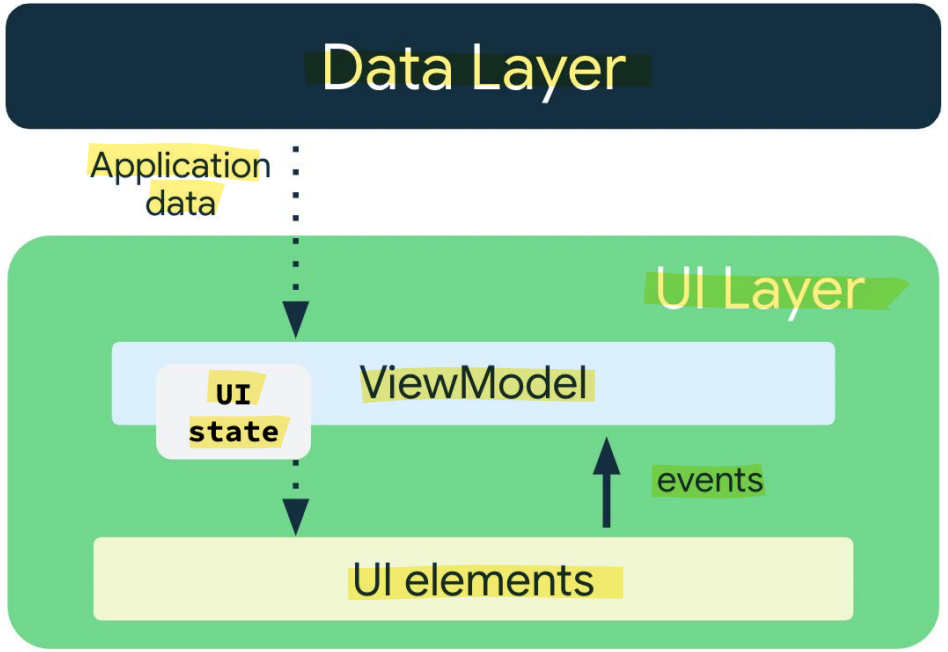


Approfondimento!

- **UI layer**
  - Il ruolo del livello UI, o livello di presentazione, è quello di visualizzare i dati dell'applicazione sullo schermo
  - Ogni volta che i dati cambiano a causa di un'interazione dell'utente, come la pressione di un pulsante, l'interfaccia utente dovrebbe aggiornarsi per riflettere le modifiche
  - Il livello dell'interfaccia utente è costituito dai seguenti componenti:
    - **UI elements** (elementi dell'interfaccia utente): componenti che visualizzano i dati sullo schermo. Costruisci questi elementi usando Jetpack Compose
    - **State holders**: componenti che contengono i dati, li espongono all'interfaccia utente e gestiscono la logica dell'app. Un esempio è ViewModel

View Model

- Il componente ViewModel contiene ed espone lo stato utilizzato dall'interfaccia utente
- Lo stato dell'interfaccia utente è costituito dai dati dell'applicazione trasformati da ViewModel
- ViewModel consente alla tua app di seguire il principio dell'architettura di «guidare» l'interfaccia utente dal modello
- ViewModel archivia i dati relativi all'app che non vengono distrutti quando l'attività viene distrutta e ricreata dal framework Android
- A differenza dell'istanza dell'activity, gli oggetti ViewModel non vengono distrutti
  - L'app conserva automaticamente gli oggetti ViewModel durante le modifiche alla configurazione in modo che i dati in essi contenuti siano immediatamente disponibili dopo la ricomposizione
  - Per implementare ViewModel nella tua app, estendi la classe ViewModel di *Architecture Components* e archivia i dati dell'app all'interno di tale classe



Vantaggi di ViewModel

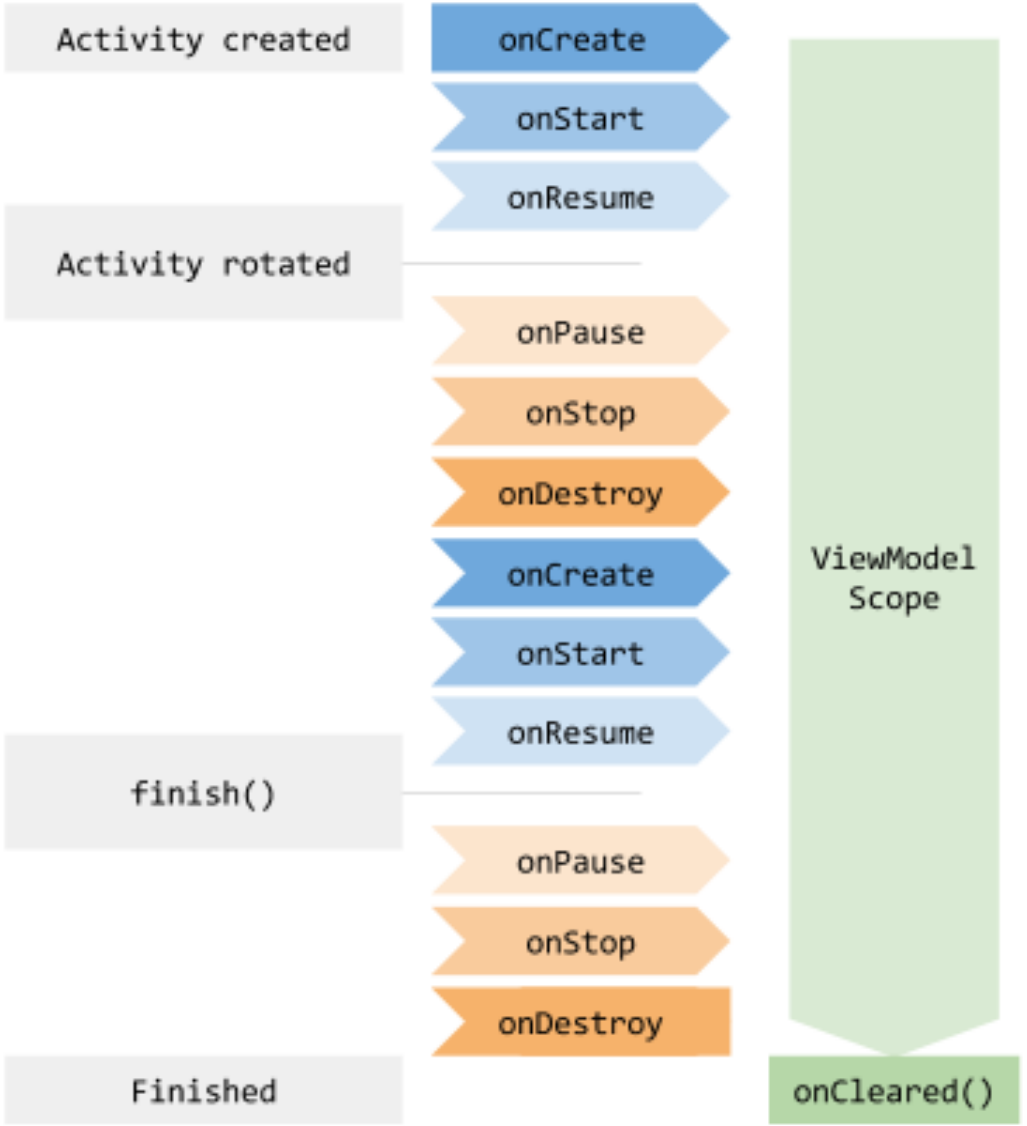
- L'alternativa a un ViewModel è una classe semplice che contiene i dati visualizzati nell'interfaccia utente
  - Questo può diventare un problema durante la navigazione tra attività o destinazioni di Navigation, se non si salvano i dati correttamente attraverso saving instance state mechanism => approccio sconsigliatissimo!
- ViewModel fornisce una comoda API per la persistenza dei dati che risolve questo problema
- I principali vantaggi della classe ViewModel sono essenzialmente due
  - Ti consente di mantenere lo **stato dell'interfaccia utente**
  - Fornisce l'accesso alla **business logic**
- Nota: ViewModel supporta completamente l'integrazione con le principali librerie Jetpack come Navigation, oltre a Compose

Persistenza

- ViewModel consente la persistenza sia attraverso
  - lo stato che detiene un ViewModel
  - Che le operazioni attivate da un ViewModel
- Questa memorizzazione nella cache significa che non è necessario recuperare nuovamente i dati attraverso modifiche di configurazione comuni, come una rotazione dello schermo
- Quando crei un'istanza di ViewModel, gli passi un oggetto che implementa l'interfaccia ViewModelStoreOwner
  - Può trattarsi di una destinazione di Navigation, di un grafico di navigazione, di un activity
- Il tuo ViewModel viene quindi «limitato» al ciclo di vita di ViewModelStoreOwner
  - Rimane in memoria finché il suo ViewModelStoreOwner non scompare definitivamente

The lifecycle of a ViewModel

- Il ciclo di vita di un ViewModel è legato direttamente al suo ambito (scope)
- Un ViewModel rimane in memoria fino a quando il ViewModelStoreOwner a cui è limitato scompare. Ciò può verificarsi nei seguenti contesti:
  - nel caso di un'activity, quando termina
  - mentre nel caso di Navigation quando viene rimosso dallo stack
- Solitamente si richiede un ViewModel la prima volta che il sistema chiama il metodo onCreate() di un oggetto Activity. Il sistema può chiamare onCreate() più volte durante la vita di un'activity, ad esempio quando viene ruotato lo schermo di un dispositivo. ViewModel esiste da quando si richiede per la prima volta un ViewModel fino al completamento e alla distruzione dell'activity



Un Composable è una funzione che costruisce la UI, ma non è progettata per essere un oggetto che può contenere e gestire lo stato in modo persistente. Un ViewModel, d'altra parte, è un componente che gestisce lo stato dell'applicazione in modo durevole e reagisce ai cambiamenti dei dati.

Un Composable non può essere un ViewModelStoreOwner, cioè non può gestire il ciclo di vita dei ViewModel come una Activity o una NavHost (in caso di navigazione). Questo perché un Composable è solo una funzione che definisce la UI, mentre un ViewModelStoreOwner è il componente che può trattenere e fornire il ViewModel.

View Model e Jetpack Compose

- Quando si utilizza Jetpack Compose, ViewModel è il mezzo principale per esporre lo stato dell'interfaccia utente dello screen ai composable
- La cosa più importante da tenere a mente quando si usa ViewModel con Compose è che non è possibile definire un ViewModel come Composable
  - Questo perché un Composable non è un ViewModelStoreOwner
    - Abbiamo detto prima che un ViewModelStoreOwner può trattarsi di una destinazione di Navigation, di un grafico di navigazione, di un activity
- Per ottenere i vantaggi di ViewModel in Compose, occorre ospitare ogni screen in una activity oppure usare Compose Navigation e usare ViewModels in funzioni composable nella destinazione di Navigation
    - Questo perché è possibile definire l'ambito di un ViewModel in destinazioni di Navigation, grafici di navigazione, activity

Per sfruttare correttamente i ViewModel in Compose, devi ospitare ciascun schermo (o composable) in un contesto che supporti ViewModelStoreOwner. Questo può essere fatto tramite uno dei seguenti approcci:
- Activity: Ogni Activity è un ViewModelStoreOwner e puoi associare un ViewModel direttamente ad essa. Così, il ViewModel rimarrà associato a quell'Activity e persistente anche durante la rotazione dello schermo.
- Navigation: Se usi Jetpack Navigation in Compose, le destinazioni di navigazione possono essere trattate come ViewModelStoreOwner. Ogni destinazione di navigazione (o schermata) può quindi avere un ViewModel separato, gestendo così lo stato specifico di ciascuna schermata durante la navigazione.

Aggiungere un ViewModel

- Apri build.gradle(Module: Unscramble), nel blocco delle dipendenze e aggiungi la seguente dipendenza per ViewModel

```
implementation "androidx.lifecycle:lifecycle-viewmodel-compose:2.5.1”
```

- Creare una classe ed estenderla con ViewModel

```
import androidx.lifecycle.ViewModel

        class MyClassViewModel : ViewModel() {
}
```

- Nel package dell'interfaccia utente, aggiungi una classe modello per l'interfaccia utente di stato denominata MyClassUiState

```
Data class MyClassUiState(
  val currentValue: String = ""
)
```

StateFlow

StateFlow è fondamentalmente un flusso che emette il valore dello stato corrente, e lo fa in modo che qualsiasi componente che ascolta il flusso riceva automaticamente gli aggiornamenti quando il valore cambia.

StateFlow è un data holder observable flow che emette gli aggiornamenti di stato correnti e nuovi

- La sua proprietà value riflette il valore dello stato corrente
- Per aggiornare lo stato e inviarlo al flusso, occorre assegnare un nuovo valore alla proprietà value della classe MutableStateFlow
- Uno StateFlow può essere esposto da MyClasseUiState in modo che i componenti Composable possano ascoltare gli aggiornamenti dello stato dell'interfaccia utente e fare in modo che lo stato dello schermo sopravviva alle modifiche della configurazione

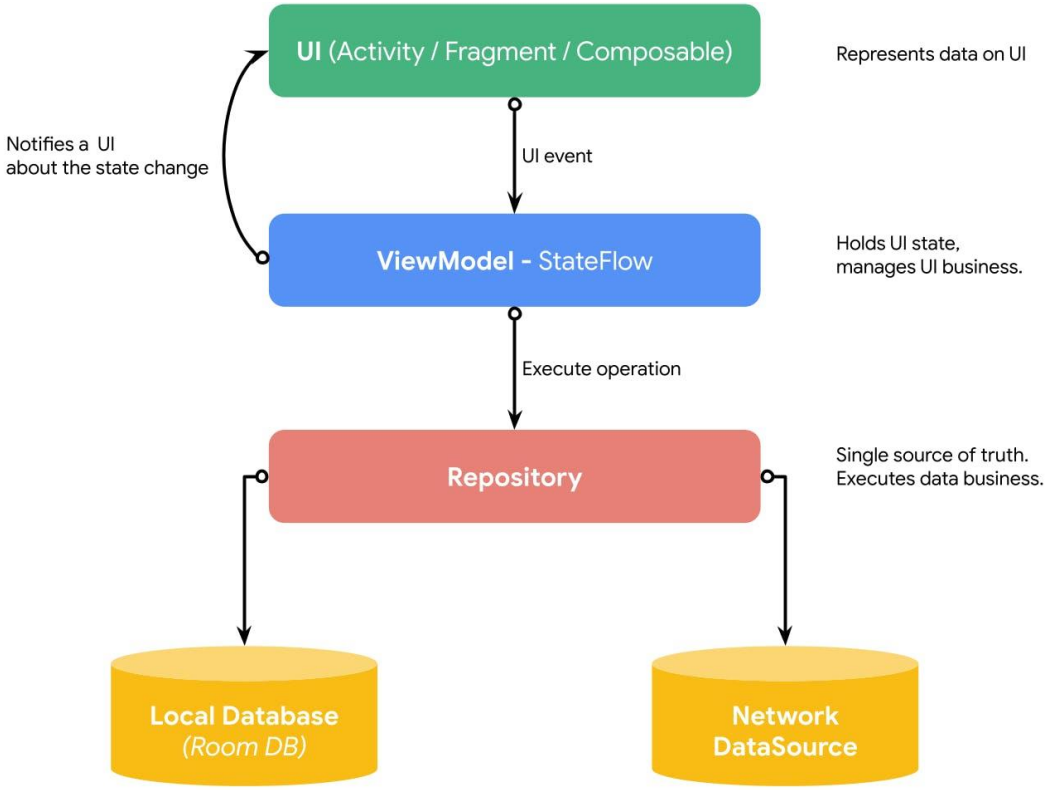
Un altro vantaggio di StateFlow in combinazione con ViewModel è che lo stato può essere mantenuto e sopravvivere a cambiamenti di configurazione, come la rotazione dello schermo. Poiché StateFlow è progettato per essere un flusso osservabile e persistente, i dati rimangono consistenti anche quando l'interfaccia utente cambia.

Esempio

```
import kotlinx.coroutines.flow.MutableStateFlow

// MyClass UI state
private val _uiState =
MutableStateFlow(MyClassUiState())
// Backing property to avoid state updates from
other classes
val uiState: StateFlow<MyClassUiState>
```

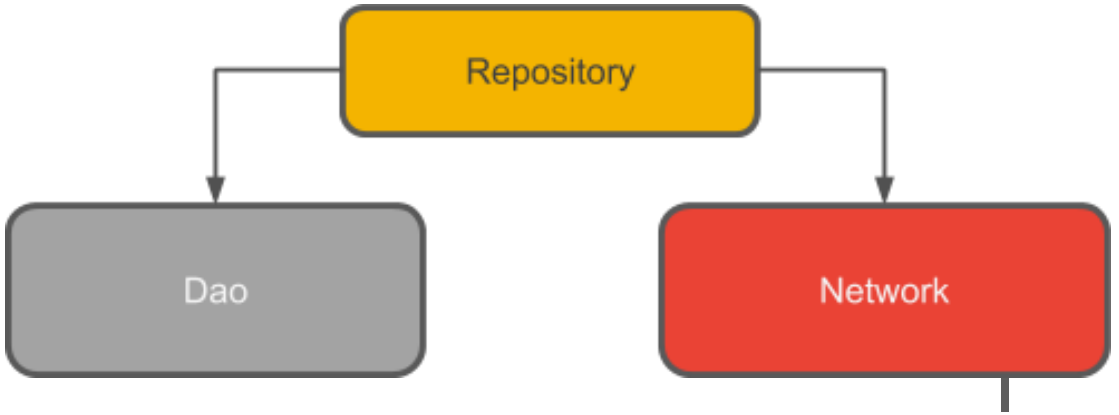
Ricapitolando



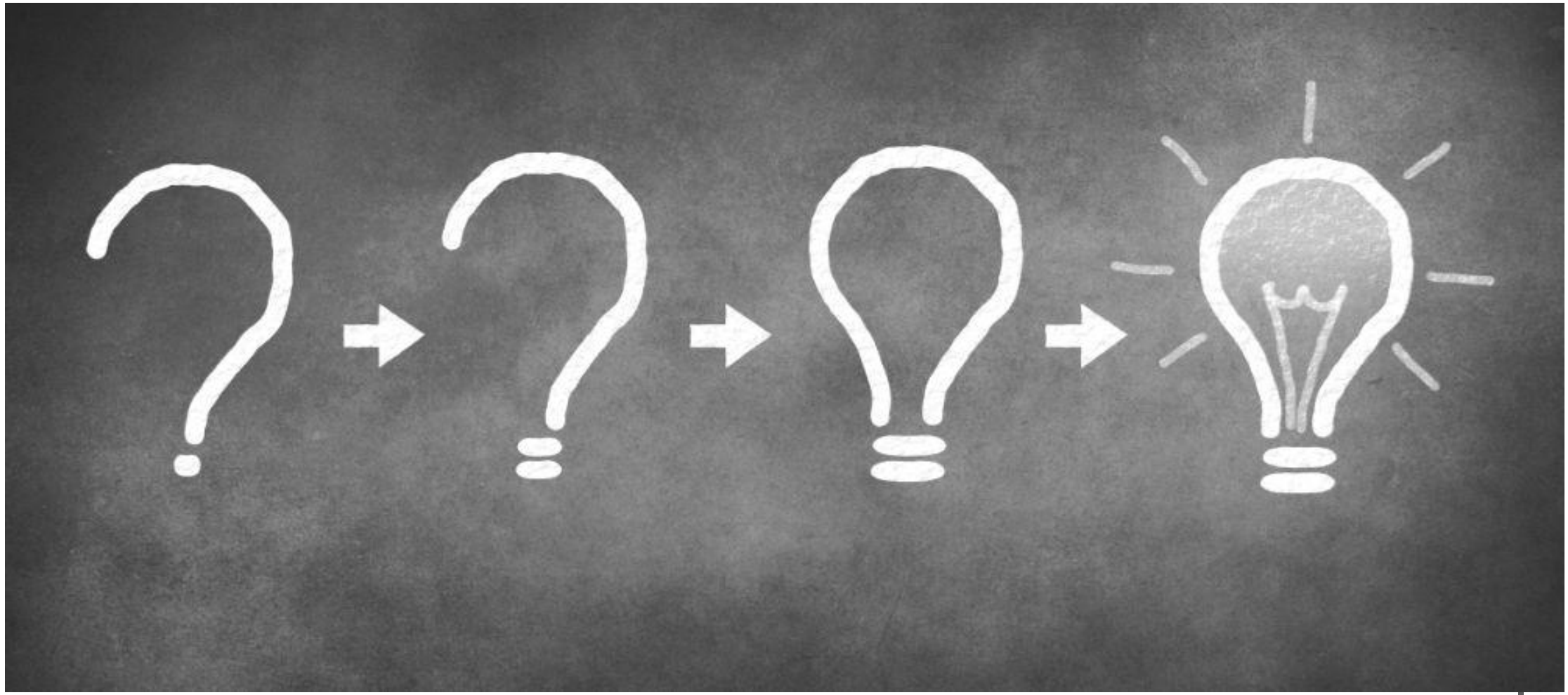
- Abbiamo appena visto ViewModel e StateFlow
- **RoomDatabase:** lo facciamo in dettaglio nella lezione 8.3
- **Repository:** utilizzato per gestire più data source

Repository

- Una classe Repository astrae l'accesso a più data source
- Repository non fa parte delle librerie di Architecture Components, ma il suo utilizzo è consigliato per la separazione del codice e l'architettura
- Una classe Repository fornisce un'API pulita per l'accesso ai dati al resto dell'applicazione



Domande???



Riferimenti e Approfondimenti

- https://developer.android.com/topic/libraries/architecture/viewmodel
- https://developer.android.com/codelabs/basic-android-kotlin-compose-viewmodel-and-state#3
- https://developer.android.com/kotlin/flow/stateflow-and-sharedflow