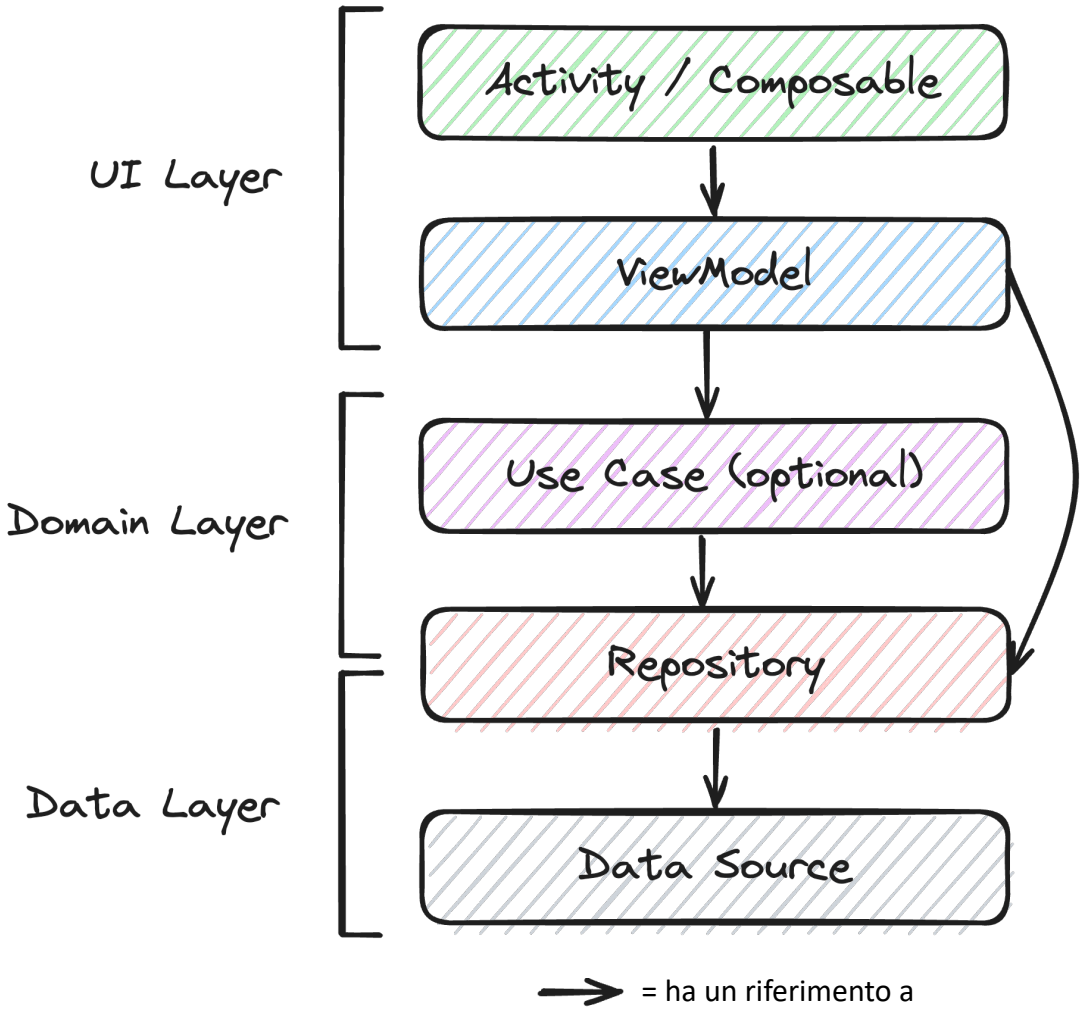
ViewModel , DataStore, Dependency Injection

Architettura delle app Compose

Laboratorio 3

Architettura di un’app Compose



Definiscono gli elementi dell’interfaccia

Tengono traccia dello stato dell’app

Definiscono parti riutilizzabili di business logic

Modellano i dati e la business logic

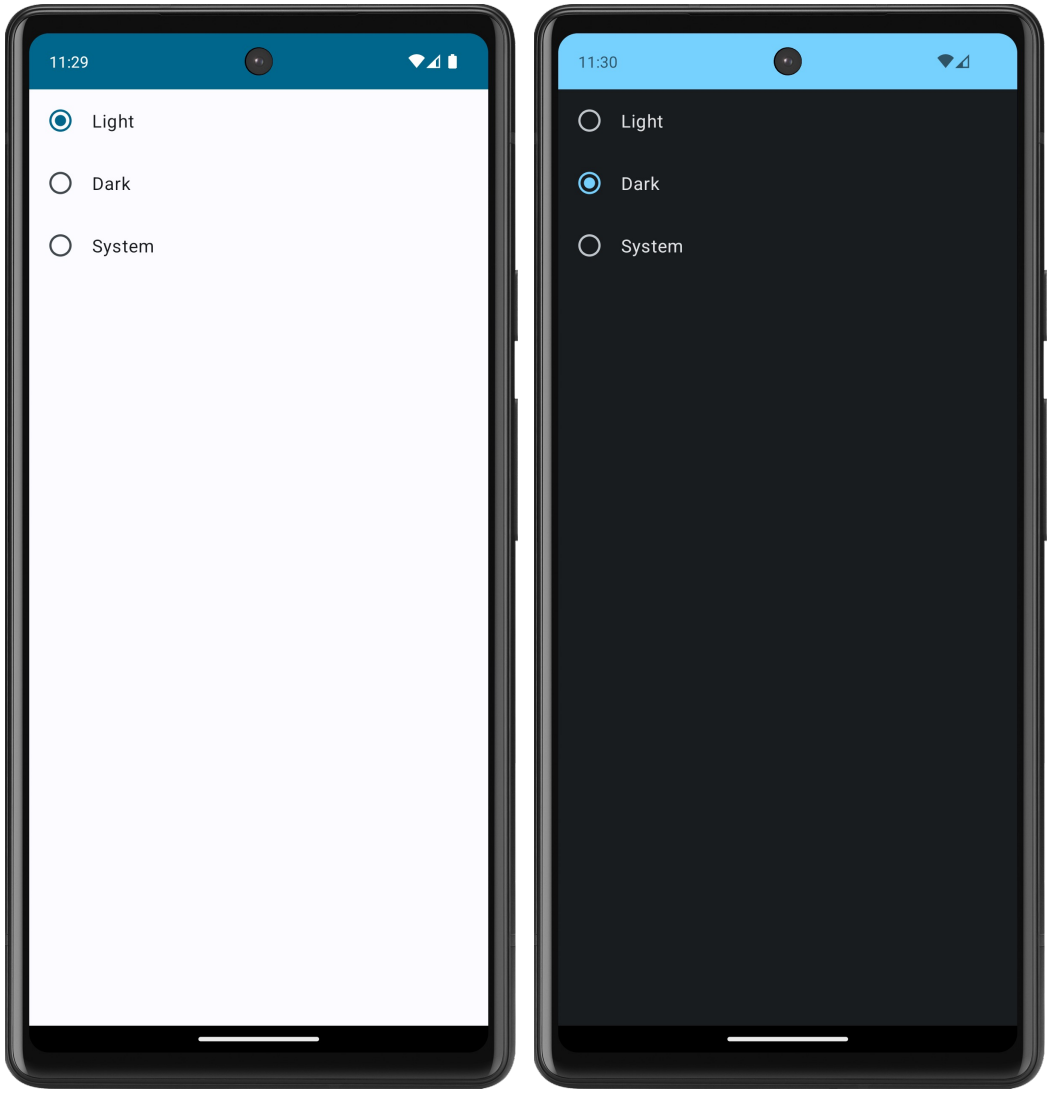
Fanno da interfaccia tra l’app e il sistema operativo per le operazioni sui dati

Laboratorio di oggi

- Esercitazione guidata
  1. Creazione di una semplice app per cambiare il tema tra chiaro, scuro e automatico
  2. Aggiunta all'app di un ViewModel per gestire lo stato dell'interfaccia
  3. Salvataggio dell'opzione selezionata tramite DataStore
     - Con dependency injection
     - Con migliore gestione delle stringhe per l'internazionalizzazione
- Esercitazione libera
  4. Aggiunta all'app TravelDiary di ViewModel, DataStore e dependency injection

1. App con toggle per il tema

- Creiamo una semplice applicazione che permetta, tramite dei radio button, di selezionare il tema tra chiaro, scuro e di sistema (che segue cioè il tema del dispositivo)
- Step
  1. Enum per rappresentare le tre opzioni
  2. Variabile per memorizzare il tema selezionato
  3. Radio buttons
  4. Applicazione del tema



1.1. Enum per rappresentare le tre opzioni

- Possiamo utilizzare un enum per definire i tre temi selezionabili

```
enum class Theme { Light, Dark, System }
```

1.2. Variabile per memorizzare il tema selezionato

```
var selectedTheme by remember { mutableStateOf(Theme.System) }
```

- Tramite **`remember`** l'app non perde traccia del tema in seguito alle recomposition
- Tramite **`mutableStateOf`** l'app fa scattare una recomposition al cambiare del valore di **`selectedTheme`**
- Scegliamo come tema di default quello di sistema

1.3. Radio buttons

```
Column(Modifier.selectableGroup()) {
    Theme.entries.forEach { theme ->
        Row(
            Modifier
                .fillMaxWidth()
                .height(56.dp)
                .selectable(
                    selected = (theme == selectedTheme),
                    onClick = { selectedTheme = theme },
                    role = Role.RadioButton
                )
                .padding(horizontal = 16.dp),
            verticalAlignment = Alignment.CenterVertically
        ) {
            RadioButton(
                selected = (theme == selectedTheme),
                onClick = null
            )
            Text(
                text = theme.toString(),
                style = MaterialTheme.typography.bodyLarge,
                modifier = Modifier.padding(start = 16.dp)
            )
        }
    }
}
```

- Codice preso dalla documentazione ufficiale e leggermente modificato

1.3. Radio buttons

```
Column(Modifier.selectableGroup()) {
    Theme.entries.forEach { theme ->
        Row(
            Modifier
                .fillMaxWidth()
                .height(56.dp)
                .selectable(
                    selected = (theme == selectedTheme),
                    onClick = { selectedTheme = theme },
                    role = Role.RadioButton
                )
                .padding(horizontal = 16.dp),
            verticalAlignment = Alignment.CenterVertically
        ) {
            RadioButton(
                selected = (theme == selectedTheme),
                onClick = null
            )
            Text(
                text = theme.toString(),
                style = MaterialTheme.typography.bodyLarge,
                modifier = Modifier.padding(start = 16.dp)
            )
        }
    }
}
```

- Codice preso dalla documentazione ufficiale e leggermente modificato

Creiamo un radio button per ogni elemento dell'enum

Definiamo quando è selezionato

Definiamo cosa succede al click

onClick = null è consigliato dalla documentazione per motivi di accessibilità

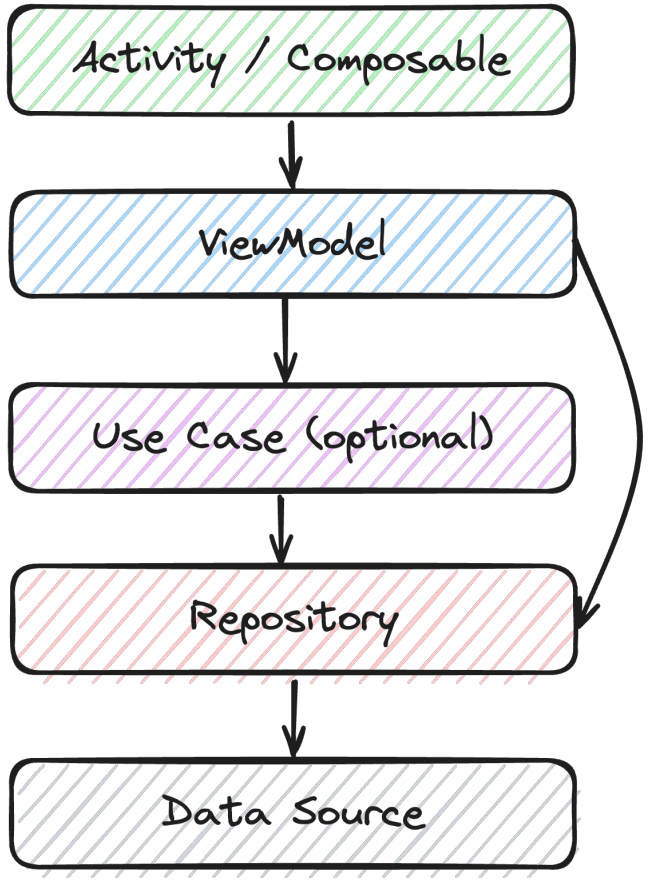
1.4. Applicazione del tema

- Nella **`MainActivity`**, impostiamo il parametro **`darkTheme`** del composable che definisce il tema

```
ThemeToggleTheme(
    darkTheme = when (selectedTheme) {
        Theme.Light -> false
        Theme.Dark -> true
        Theme.System -> isSystemInDarkTheme()
    }
) {
    // A surface container using the 'background' color from the theme
    Surface(
```

2. Aggiunta ViewModel

- L'app che abbiamo creato è semplice e funzionale
- Tuttavia, con l'implementazione di nuove feature sarà sempre più complicata da manutenere e testare
  - Interfaccia, stato e dati coesistono nello stesso file (e all'interno delle stesse funzioni!)
- **Soluzione**: definiamo una struttura con una migliore *separation of concerns*
  - Il primo passaggio è l'aggiunta di un **ViewModel**



2. Aggiunta ViewModel

- Step
  1. Installazione dipendenze
  2. Struttura progetto
  3. Creazione classe ViewModel
  4. Schermata dell’app
  5. Creazione istanza ViewModel

Recap: ViewModel

- Incapsula ed espone lo stato dell'interfaccia utente di una parte dell'app (es. un'Activity, uno screen, ...)
- Espone metodi per modificare lo stato
- Sopravvive ai cambi di configurazione del dispositivo (es. rotazione)
- Il suo ciclo di vita è legato automaticamente a quello del suo owner (es. un'Activity, una parte del grafo di navigazione, ...)

Recap: StateFlow

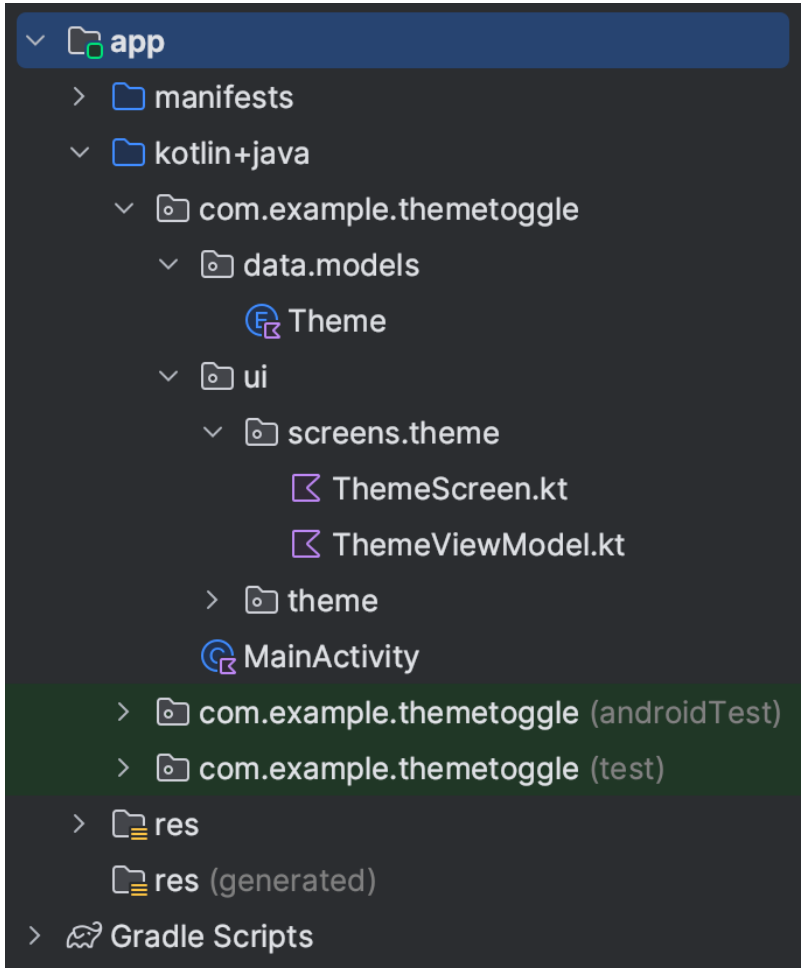
- È un observable con una singola proprietà **`value`**, che può essere modificata facendo scattare un'aggiornamento a tutti i subscriber
- È il metodo più indicato per tracciare i cambiamenti di stato all'interno di un ViewModel
- Fornisce vari operatori per l'aggiornamento di **`value`**, tutti atomici e utilizzabili in contesti concorrenti e multi-thread

2.1. Installazione dipendenze

- Aggiungere le seguenti dipendenze al file **`build.gradle.kts`** (modulo :app)

```
implementation("androidx.lifecycle:lifecycle-viewmodel-compose:2.7.0")
implementation("androidx.lifecycle:lifecycle-runtime-compose:2.7.0")
```

2.2. Struttura progetto



2.3. Creazione classe ViewModel

ThemeViewModel.kt

```
data class ThemeState(val theme: Theme)

class ThemeViewModel : ViewModel() {
    private val _state =
        MutableStateFlow(ThemeState(Theme.System))
    val state = _state.asStateFlow()

    fun changeTheme(theme: Theme) {
        _state.value = ThemeState(theme)
    }
}
```

- È convenzione raggruppare lo stato del ViewModel in una classe State
- Utilizziamo un **`MutableStateFlow`** come backing property per lo stato, così che possa essere modificato internamente al ViewModel, mentre esponiamo uno **`StateFlow`** non modificabile esternamente
- Esponiamo un metodo **`changeTheme`** per cambiare tema

2.4. Schermata dell’app

ThemeScreen.kt

```
@Composable
fun ThemeScreen(state: ThemeState, onThemeSelected: (theme: Theme) -> Unit) {
    Column(Modifier.selectableGroup()) {
        Theme.entries.forEach { theme ->
            Row(
                Modifier
                    .fillMaxWidth()
                    .height(56.dp)
                    .selectable(
                        selected = (theme == state.theme),
                        onClick = { onThemeSelected(theme) },
                        role = Role.RadioButton
                    )
                    .padding(horizontal = 16.dp),
                verticalAlignment = Alignment.CenterVertically
            ) {
                RadioButton(selected = (theme == state.theme), onClick = null)
                Text(
                    text = theme.toString(),
                    style = MaterialTheme.typography.bodyLarge,
                    modifier = Modifier.padding(start = 16.dp)
                )
            }
        }
    }
}
```

2.4. Schermata dell'app

ThemeScreen.kt

```
@Composable
fun ThemeScreen(state: ThemeState, onThemeSelected: (theme: Theme) -> Unit) {
    Column(Modifier.selectableGroup()) {
        Theme.entries.forEach { theme ->
            Row(
                Modifier
                    .fillMaxWidth()
                    .height(56.dp)
                    .selectable(
                        selected = (theme == state.theme),
                        onClick = { onThemeSelected(theme) },
                        role = Role.RadioButton
                    )
                    .padding(horizontal = 16.dp),
                verticalAlignment = Alignment.CenterVertically
            ) {
                RadioButton(selected = (theme == state.theme), onClick = null)
                Text(
                    text = theme.toString(),
                    style = MaterialTheme.typography.bodyLarge,
                    modifier = Modifier.padding(start = 16.dp)
                )
            }
        }
    }
}
```

Riceviamo come parametri:
- lo stato del ViewModel
- la funzione da richiamare alla modifica dello stato

2.5. Creazione istanza ViewModel

MainActivity.kt

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            val themeViewModel = viewModel<ThemeViewModel>()
            val themeState by themeViewModel.state.collectAsStateWithLifecycle()

            ThemeToggleTheme(
                darkTheme = when (themeState.theme) {
                    Theme.Light -> false
                    Theme.Dark -> true
                    Theme.System -> isSystemInDarkTheme()
                }
            ) {
                // A surface container using the 'background' color from the theme
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    ThemeScreen(themeState, themeViewModel::changeTheme)
                }
            }
        }
    }
}
```

2.5. Creazione istanza ViewModel

MainActivity.kt

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            val themeViewModel = viewModel<ThemeViewModel>()
            val themeState by themeViewModel.state.collectAsStateWithLifecycle()

            ThemeToggleTheme(
                darkTheme = when (themeState.theme) {
                    Theme.Light -> false
                    Theme.Dark -> true
                    Theme.System -> isSystemInDarkTheme()
                }
            ) {
                // A surface container using the 'background' color from the theme
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    ThemeScreen(themeState, themeViewModel::changeTheme)
                }
            }
        }
    }
}
```

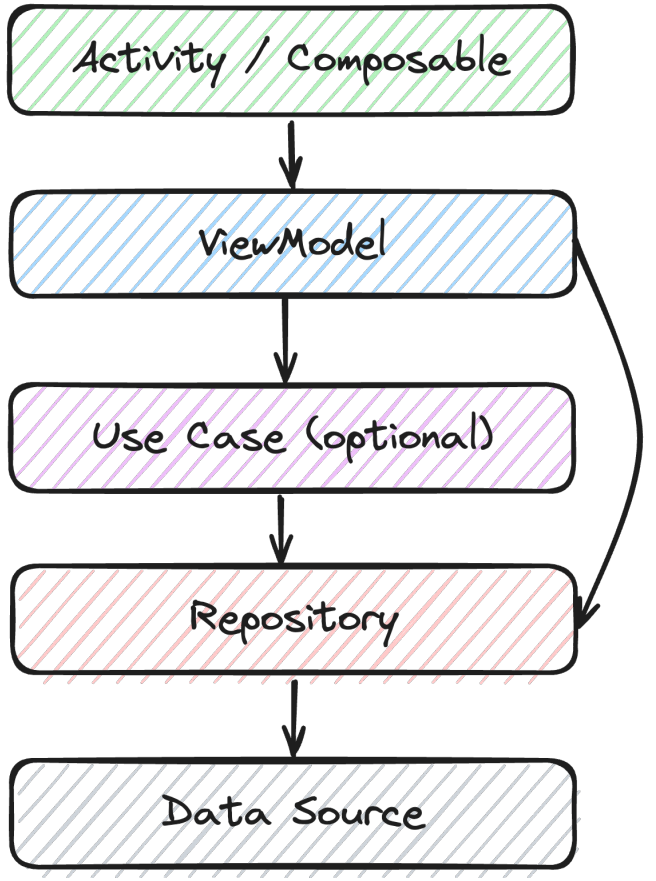
Creiamo l'istanza tramite l'helper viewModel()

Convertiamo lo StateFlow in State

Passiamo lo state alla schermata, assieme alla funzione per modificare il tema

3. Aggiunta DataStore

- All'app che abbiamo creato manca una feature importante: il tema selezionato viene salvato solo in memory
  - Quindi va perso alla chiusura dell'app!
- Aggiungiamo un data layer tramite cui memorizzare il tema sul dispositivo
  - Tramite **DataStore**



3. Aggiunta DataStore

- Step
  1. Installazione dipendenze per DataStore
  2. Creazione repository
  3. Aggiunta della repository al ViewModel
  4. Installazione dipendenze per dependency injection con Koin
  5. Creazione AppModule
  6. Avvio di Koin
  7. Creazione del ViewModel tramite Koin

Recap: DataStore

- È una soluzione per lo storage di dati che permette di salvare coppie chiave-valore o oggetti tipizzati
- È perfettamente integrato nell'ecosistema di Compose, utilizzando i Flow e le coroutine di Kotlin
- Fornsce due API:
  - **Preferences DataStore**: per il salvataggio di coppie chiave-valore (lo utilizzeremo ora)
  - **Proto DataStore**: per memorizzare tipi di dati custom

3.1. Installazione dipendenze per DataStore

- Aggiungere la seguente dipendenza al file **`build.gradle.kts`** (modulo :app)

```
implementation("androidx.datastore:datastore-preferences:1.0.0")
```

3.2. Creazione repository

data.repositories.ThemeRepository.kt

```
class ThemeRepository(private val dataStore: DataStore<Preferences>) {
    companion object {
        private val THEME_KEY = stringPreferencesKey("theme")
    }

    val theme = dataStore.data
        .map { preferences ->
            try {
                Theme.valueOf(preferences[THEME_KEY] ?: "System")
            } catch (_: Exception) {
                Theme.System
            }
        }

    suspend fun setTheme(theme: Theme) =
        dataStore.edit { it[THEME_KEY] = theme.toString() }
}
```

3.2. Creazione repository

data.repositories.ThemeRepository.kt

```
class ThemeRepository(private val dataStore: DataStore<Preferences>) {
    companion object {
        private val THEME_KEY = stringPreferencesKey("theme")
    }

    val theme = dataStore.data
        .map { preferences ->
            try {
                Theme.valueOf(preferences[THEME_KEY] ?: "System")
            } catch (_: Exception) {
                Theme.System
            }
        }

    suspend fun setTheme(theme: Theme) =
        dataStore.edit { it[THEME_KEY] = theme.toString() }
}
```

Accettiamo il DataStore come parametro

Sottoscriviamo al Flow del DataStore

Convertiamo la stringa del tema in enum

Se la conversione produce errori (es. la stringa non corrisponde a un valore dell'enum) utilizziamo il tema di sistema

3.2. Creazione repository

data.repositories.ThemeRepository.kt

```
class ThemeRepository(private val dataStore: DataStore<Preferences>) {
    companion object {
        private val THEME_KEY = stringPreferencesKey("theme")
    }

    val theme = dataStore.data
        .map { preferences ->
            try {
                Theme.valueOf(preferences[THEME_KEY] ?: "System")
            } catch (_: Exception) {
                Theme.System
            }
        }

    suspend fun setTheme(theme: Theme) =
        dataStore.edit { it[THEME_KEY] = theme.toString() }
}
```

Definiamo una suspend function per modificare il valore nello store da una coroutine

3.3. Aggiunta repository al ViewModel

- Modifichiamo il ViewModel per leggere e scrivere i dati dalla repository

```
data class ThemeState(val theme: Theme)

class ThemeViewModel(
    private val repository: ThemeRepository
) : ViewModel() {
    val state = repository.theme.map { ThemeState(it) }.stateIn(
        scope = viewModelScope,
        started = SharingStarted.WhileSubscribed(),
        initialValue = ThemeState(Theme.System)
    )

    fun changeTheme(theme: Theme) = viewModelScope.launch {
        repository.setTheme(theme)
    }
}
```

3.3. Aggiunta repository al ViewModel

- Modifichiamo il ViewModel per leggere e scrivere i dati dalla repository

```
data class ThemeState(val theme: Theme)

class ThemeViewModel(
    private val repository: ThemeRepository
) : ViewModel() {
    val state = repository.theme.map { ThemeState(it) }.stateIn(
        scope = viewModelScope,
        started = SharingStarted.WhileSubscribed(),
        initialValue = ThemeState(Theme.System)
    )

    fun changeTheme(theme: Theme) = viewModelScope.launch {
        repository.setTheme(theme)
    }
}
```

Accettiamo la repository come parametro

Convertiamo il Flow del tema in uno StateFlow di ThemeState

Utilizziamo una coroutine nello scope del ViewModel per modificare il tema nella repository

Dependency injection

- **Problema**: come passiamo i parametri necessari ai costruttori di ViewModel (che necessita la repository) e repository (a cui serve il DataStore)?
- Potremmo farlo manualmente, ma:
  - Le istanze passate sarebbero dei singleton?
  - Sarebbe semplice fare refactoring del codice?
  - Sarebbe semplice testare separatamente i vari componenti?
  - ...
- **Soluzione**: **dependency injection**

Dependency injection

- La **dependency injection** (**DI**) è una tecnica molto utilizzata che permette a un sistema di avere una buona architettura, fornendo i seguenti vantaggi:
  - Riusabilità del codice
  - Facilità di refactoring
  - Facilità di testing

Dependency injection

- Le classi spesso richiedono riferimenti ad altre classi. Per esempio, una classe **`Car`** potrebbe aver bisogno di un riferimento a una classe **`Engine`**. Queste classi necessarie sono chiamate *dipendenze* e, in questo esempio, la classe **`Car`** dipende dalla presenza di un'istanza della classe **`Engine`** per funzionare
- Una classe può ottenere un oggetto di cui ha bisogno in tre modi:
  a. La classe stessa crea un'istanza della dipendenza di cui ha bisogno
  b. La classe ottiene un riferimento alla dipendenza da qualche altra parte, tramite API come **`Context`** e **`getSystemService`**
  c. La dipendenza viene fornita come parametro. L'applicazione può passare queste dipendenze quando la classe viene istanziata o passarle ai metodi della classe che ne hanno bisogno

Dependency injection

- Le classi spesso richiedono riferimenti ad altre classi. Per esempio, una classe **`Car`** potrebbe aver bisogno di un riferimento a una classe **`Engine`**. Queste classi necessarie sono chiamate *dipendenze* e, in questo esempio, la classe **`Car`** dipende dalla presenza di un'istanza della classe **`Engine`** per funzionare
- Una classe può ottenere un oggetto di cui ha bisogno in tre modi:
  a. La classe stessa crea un'istanza della dipendenza di cui ha bisogno
  b. La classe ottiene un riferimento alla dipendenza da qualche altra parte, tramite API come **`Context`** e **`getSystemService`**
  c. La dipendenza viene fornita come parametro. L'applicazione può passare queste dipendenze quando la classe viene istanziata o passarle ai metodi della classe che ne hanno bisogno

Dependency injection!

Dependency injection con Koin

- **Koin** è una libreria open source, semplice e modulare per la gestione automatica della dependency injection
- Non è specifica per Android, può funzionare praticamente in qualsiasi progetto Kotlin
- Tramite Koin, è possibile organizzare le dipendenze della nostra app in **moduli**
- Ogni modulo può definire varie tipologie di oggetto. Nel nostro caso utilizzeremo:
  - **`single`** per la creazione di singleton
  - **`viewModel`** per istanziare appunto i ViewModel Android

3.4. Installazione dipendenze per DI

- Aggiungere la seguente dipendenza al file **`build.gradle.kts`** (modulo :app)

```
implementation("io.insert-koin:koin-androidx-compose:3.5.3")
```

3.5. Creazione AppModule

AppModule.kt

```
val Context.dataStore
    by preferencesDataStore("theme")

val appModule = module {
    single { get<Context>().dataStore }

    single { ThemeRepository(get()) }

    viewModel { ThemeViewModel(get()) }
}
```

- Il DataStore va creato come extension di **`Context`**, di conseguenza dobbiamo inizializzarlo top-level
- Con la funzione **`get()`** possiamo dire a Koin di andare a cercare la dipendenza in questione all'interno dei suoi moduli
- Ma come fa Koin a reperire un'istanza di **`Context`**?
  - Gli viene fornita in fase di avvio dell'app (vedi prossima slide)

3.6. Avvio di Koin

- La DI di Koin va avviata a livello globale di applicazione
- Per farlo, possiamo creare una classe nella root del progetto che eredita da **`Application`**, e fare override del metodo **`onCreate`**
- La classe va aggiunta al file manifest

ThemeToggleApplication.kt

```
class ThemeToggleApplication : Application() {
    override fun onCreate() {
        super.onCreate()

        startKoin {
            androidLogger()
            androidContext(this@ThemeToggleApplication)
            modules(appModule)
        }
    }
}
```

AndroidManifest.xml

```
<application
    android:name=".ThemeToggleApplication"
```

3.7. Creazione ViewModel tramite Koin

- L'ultimo passaggio è quello di sostituire l'helper **`viewModel`** con quello fornito da Koin per istanziare il ViewModel

```
val themeViewModel = viewModel<ThemeViewModel>()
```

```
val themeViewModel = koinViewModel<ThemeViewModel>()
```

3.8. Bonus: gestione delle stringhe e internazionalizzazione

- Possiamo gestire meglio le stringhe all'interno della nostra app
- E aggiungere il supporto multilingua!

3.8. Bonus: gestione delle stringhe e internazionalizzazione

1. Nel file **`res/values/strings.xml`**, creiamo una entry per tutte le stringhe dell'interfaccia utente

```
<string name="theme_light">Light</string>
<string name="theme_dark">Dark</string>
<string name="theme_system">System</string>
```

3.8. Bonus: gestione delle stringhe e internazionalizzazione

2. Utilizziamo i valori definiti nel file all'interno dell'app (ThemeScreen.kt)

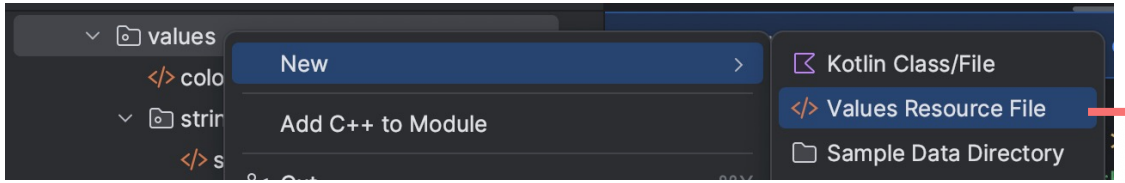
```
text = theme.toString(),
```

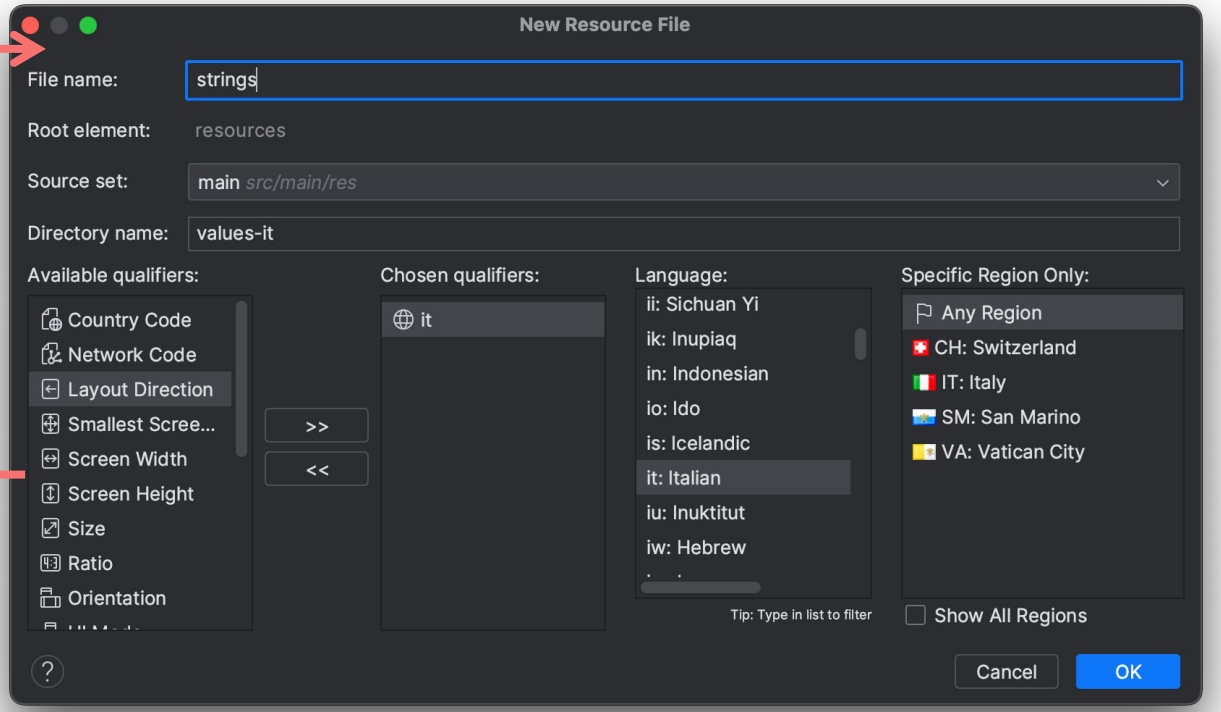


```
text = stringResource(when (theme) {
    Theme.Light -> R.string.theme_light
    Theme.Dark -> R.string.theme_dark
    Theme.System -> R.string.theme_system
}),
```

3.8. Bonus: gestione delle stringhe e internazionalizzazione

3. Aggiungiamo un file per le stringhe in italiano






```
<string name="theme_light">Chiaro</string>
<string name="theme_dark">Scuro</string>
<string name="theme_system">Tema di Sistema</string>
```

3.8. Bonus: gestione delle stringhe e internazionalizzazione

- Cambiando la lingua del dispositivo in italiano, cambierà anche il testo all'interno della nostra app
  - Settings → System → Languages → System Languages → Add a Language
  - Selezionare "Italiano (Italia)" e trascinarlo sopra all'inglese

4. TravelDiary - Struttura avanzata progetto

- Modificare l'architettura dell'app TravelDiary in base a quanto visto negli esercizi di oggi:
  - Configurare la dependency injection con Koin
  - Aggiungere il supporto ai ViewModel
    - **`AddTravelViewModel`**
    - **`SettingsViewModel`**
  - Memorizzare l'username della pagina Settings nel DataStore

4. TravelDiary - Struttura avanzata progetto

- **Hint**: navigation e creazione ViewModel
  - Il punto migliore per la creazione del ViewModel di una schermata è il blocco **`composable`** della rotta corrispondente

```
with(TravelDiaryRoute.Settings) {
    composable(route) {
        val settingsVm = koinViewModel<SettingsViewModel>()
        SettingsScreen(settingsVm.state, settingsVm::setUsername)
    }
}
```

  - Non dimenticate **`.collectAsStateWithLifecycle()`** se lo stato del ViewModel è uno **`StateFlow`** (non lo è nell'esempio sopra)

4. TravelDiary - Struttura avanzata progetto

- **Hint**: ViewModel con molti metodi
  - In base all'implementazione, **`AddTravelViewModel`** potrebbe contenere vari metodi, ad esempio: **`setDestination`**, **`setDate`**, **`setDescription`**
  - In questi casi, può essere una buona idea raggruppare tutti i metodi in un oggetto **`actions`** all'interno del ViewModel, così da non doverli passare individualmente dal ViewModel allo screen

AddTravelViewModel.kt

```
interface AddTravelActions {
    fun setDestination(title: String)
    fun setDate(date: String)
    fun setDescription(description: String)
}
```

Navigation.kt

```
AddTravelScreen(
    state,
    addTravelVm.actions,
    navController
)
```

4. TravelDiary - Struttura avanzata progetto

- **Hint**: TextField e Flow (schermata Settings)
  - Modificare il valore contenuto in un TextField mentre l'utente sta scrivendo resetta la posizione del cursore, creando una bad user experience
  - Per questo motivo, il valore di un TextField non dovrebbe mai provenire da un Flow, che può emettere nuovi valori in qualsiasi momento

4. TravelDiary - Struttura avanzata progetto

- **Hint**: TextField e Flow (schermata Settings) pt. 2
  - Come facciamo allora a sincronizzare il valore di un TextField con il DataStore?
    - Alla creazione del ViewModel, leggiamo il valore presente nel DataStore e lo salviamo in una variabile non Flow (es. **`mutableStateOf`**)
    - Ogni volta che il valore della variabile viene modificato, lo scriviamo nel DataStore
  - Come eseguiamo codice alla creazione del ViewModel?
    - Con il blocco **`init`** di Kotlin

```
class SettingsViewModel (private val repository: SettingsRepository) : ViewModel() {

    // ...

    init {
        println("SettingsViewModel created")
    }
}
```

Tocca a voi!

Riferimenti

- State in Compose
  https://developer.android.com/jetpack/compose/state
- ViewModel
  https://developer.android.com/topic/libraries/architecture/viewmodel
- StateFlow
  https://developer.android.com/kotlin/flow/stateflow-and-sharedflow
- Dependency Injection
  https://developer.android.com/training/dependency-injection
- DataStore
  https://developer.android.com/topic/libraries/architecture/datastore
- Koin
  https://insert-koin.io/