UNIVERSITA' DEGLI STUDI DI PADOVA

June 12, 2021

JETPACK COMPOSE

Ceccon Gioele, Pastore Alessandra, Pozzer Matteo

Contents

1 Introduzione

Annunciato da Google nel 2019 ed entrato in beta nel febbraio 2021, Jetpack Compose si propone di facilitare e migliorare il processo di sviluppo di applicazioni Android.

Si tratta di una novità importante, in quanto dopo aver introdotto android 1.0 nel 2010, il team di Google non ha mai veramente modificato il funzionamento iniziale del toolkit per la UI, ma si è limitato a risolvere bug e aggiungere feature. Il vecchio toolkit, dunque, era stato progettato per funzionare su dispositivi aventi schermi di dimensioni minori e montanti CPU e GPU aventi potenza inferiore rispetto ai device attuali.

(a) *S21*

(b) *S III*

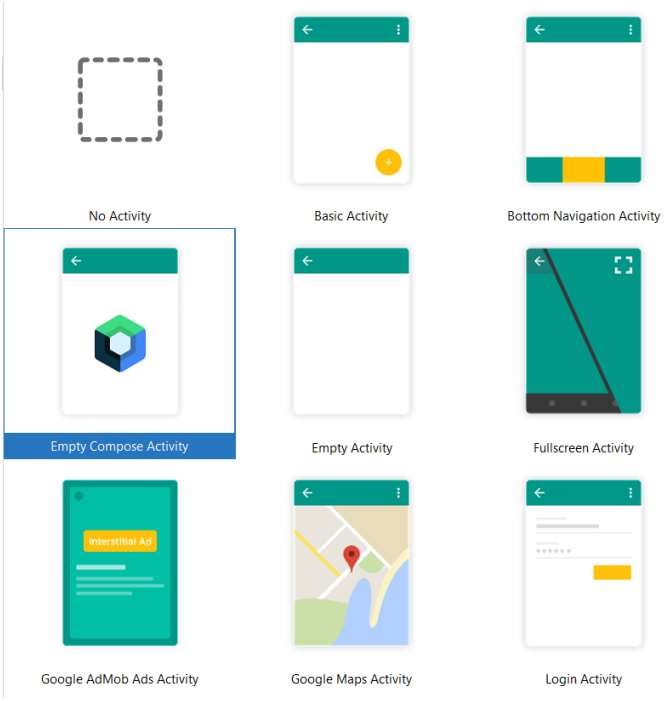
Inoltre, negli ultimi anni sono cresciute sia l'audience attirata da Android che la comunità di sviluppatori, costretti a lavorare aspettando il rilascio di nuove API per ottenere aggiornamenti. Proprio per questo motivo Google ha dunque deciso di rendere il nuovo toolkit una "libreria di supporto" ed inserirlo in Jetpack.

1.1 Caratteristiche generali

Ispirato ad altri framework come React, Litho, Vue.js e Flutter, Jetpack Compose rinuncia all'utilizzo dei file XML e adotta un approccio completamente dichiarativo per la realizzazione dell'interfaccia utente, implementata attraverso l'utilizzo di un tipo specifico di funzioni, dette composable.
In particolare, il toolkit si concentra sulla gestione dello stato, con lo scopo di renderne l'aggiornamento più semplice e al tempo stesso di ridurre la probabilità di creare bug durante questa operazione. Inoltre, Jetpack Compose propone una riduzione del numero di righe di codice attraverso la completa sinergia con Kotlin, utilizzato per gestire al meglio il flusso di dati grazie alla compattezza e maneggiabilità di questo linguaggio di programmazione. In ultima istanza, Jetpack Compose risulta essere interoperabile con altri toolkit. Dunque, oltre a progettare applicazione native Compose, è possibile aggiungere nuove funzionalità a quelle esistenti, senza l'obbligo di dover riscrivere tutto il codice.

2 Android Studio Canary

Android Studio Canary rappresenta una variante di Android Studio da usare quando è necessario utilizzare nuove tecnologie o plugin messi a disposizione da Google. Permette infatti di usufruire di nuove funzionalità di programmazione, non disponibili nella versione standard. Dunque, anche il toolkit Jetpack Compose, essendo ancora in via di sviluppo e utilizzabile attualmente nella sua versione beta, necessita di Android Studio Canary per poter essere utilizzato all'interno della propria applicazione. Questa variante dell'ambiente di sviluppo mette a disposizione delle novità apposite per Compose, come ad esempio la possibilità di mostrare la preview di certe parti dell'interfaccia utente ed un nuovo template, che prende il nome di Empty Compose Activity.



Le differenze principali tra Android Studio Canary e la versione standard sono illustrate nella tabella seguente:

Canary	**Stabile**
Gli aggiornamenti vengono rilasciati con cadenza settimanale.	Il software riceve un upgrade una volta al mese.
La versione Canary implementa quasi subito le nuove tecnologie. Ciò significa che si ha la possibilità di testare tutte le nuove feature non appena vengono rilasciate.	Nella versione stabile le nuove funzionalità vengono utilizzate solo dopo essere state adeguatamente testate.
Utilizzando questa particolare variante gli sviluppatori potrebbero incontrare qualche malfunzionamento a causa degli aggiornamenti settimanali delle nuove tecnologie.	Durante lo sviluppo di applicazioni è molto raro che la piattaforma presenti qualche anomalia.
Le applicazioni create con questa versione di android studio non sono particolarmente indicate per l'uso professionale, in quanto potrebbero essere soggette a molti bug.	Invece, la versione stabile è consigliata per la realizzazione di applicazioni professionali.

3 Compose OverView

Per poter usufruire di Jetpack Compose all'interno della propria applicazione il livello minimo di API richiesto è 21. Inoltre, è necessaria l'aggiunta del repository Maven all'interno del progetto, in quanto senza di esso non è possibile aggiungere la dipendenza a Compose. La versione minima di kotlin dell'applicazione deve essere 1.4.32.

È essenziale anche aggiungere l'estensione del compilatore Kotlin per Compose:

```
composeOptions {
    kotlinCompilerExtensionVersion "1.0.0-beta06"
    kotlinCompilerVersion '1.4.32'
}
```

Il seguente codice, posto all'interno del file build.gradle, serve per abilitare JetpackCompose all'interno di Android Studio Canary:

```
buildFeatures {
    compose true
}
```

Il toolkit Jetpack Compose è costituito da 6 blocchi principali:

- Compose.complier: ha il compito di trasformare le funzioni composable e di ottimizzare il codice sfruttando il plugin del compilatore Kotlin. Infatti, l'annotazione @Composable è referita al plugin del compilatore kotlin, e non al processore.

```
implementation "androidx.compose.compiler:compiler:1.0.0-beta06"
```

- Compose.runtime: contiene i modelli fondamentali per la programmazione tramite Compose. Al tempo stesso detiene degli Adapter per librerie definite da terzi, come ad esempio Observable, Flow o LiveData.

```
implementation "androidx.compose.runtime:runtime-livedata:$compose_version"
```

- Compose.foundation: libreria che realizza blocchi predefiniti per l'interfaccia utente pronti all'uso, come ad esempio LazyColumn.

```
import androidx.compose.foundation.lazy.LazyColumn
```

- Compose.ui: Contiene le componenti principali per realizzare l'interfaccia utente.

```
implementation "androidx.compose.ui:ui:1.0.0-beta06"
```

- Compose.animation: permette la creazione di animazioni per l'interfaccia utente.

```
implementation "androidx.compose.animation:animation:1.0.0-beta06"
```

- Compose.material: grazie ad essa vi è la possibilità di realizzare oggetti per l'interfaccia utente che includono il material Design.

```
implementation "androidx.compose.material:material:1.0.0-beta06"
```

4 Principi fondamentali

4.1 4.1 Funzioni Composable

Per definizione, sono considerate funzioni composable solo quelle funzioni che generano direttamente elementi della UI.
Si tratta di funzioni che ricevono dei parametri, i dati che caratterizzano lo stato dell'interfaccia, ed emettono elementi della gerarchia della UI in funzione di essi. Difatti, queste funzioni ritornano Unit: il loro scopo non è quello di produrre widget, ma di generare l'interfaccia utilizzando le informazioni contenute dai parametri. Per fare ciò, le funzioni composable richiamano al loro interno altre funzioni composable, siano esse funzioni base fornite da Compose o funzioni definite dallo sviluppatore: solo una funzione composable può invocare un'altra funzione composable.
A livello di codice, sono identificate dall'annotazione @Composable, che le precede. Grazie a questa annotazione, il compilatore riesce a produrre il codice necessario all'emissione degli elementi della UI. Lo sviluppatore, dunque, deve preoccuparsi di descrivere la forma, il contenuto dell'interfaccia, mentre la generazione della stessa è affidata a Jetpack Compose.

Nell'esempio sottostante, sono definite due funzioni composable:

- Buongiorno, che attraverso l'invocazione di Text, funzione composable fornita da JetPack Compose, proietta sullo schermo un testo
- Saluto, che richiama Buongiorno fornendole il parametro necessario ed inserendola tramite una lambda all'interno di una Surface, funzione composable fornita dal toolkit esattamente come Text

```
@Composable
fun Saluto(parola: String){
    Surface(color = Color.Yellow) {
        Buongiorno(parola)
    }
}

@Composable
fun Buongiorno(nome: String){
    Text( text: "Buongiorno $nome!")
}
```

Attraverso l'implementazione di una funzione composable particolare, marcata dall'annotazione @Preview, è possibile visualizzare ciò che verrà emesso

da una determinata funzione senza il bisogno di dover utilizzare l'emulatore di Android Studio.
Nel caso in esempio:

```
@Preview
@Composable
fun SalutoPreview()
{
    Saluto( parola: "JetPack Compose")
}
```

4.2 Composition e Recomposition

Jetpack Compose risolve il problema della gestione degli elementi della UI e del loro aggiornamento attraverso Composition e Recomposition.
Con il termine Composition si intende la struttura ad albero delle funzioni composable che descrivono una UI in un determinato stato.
Quando l'applicazione viene lanciata per la prima volta, viene generata la Initial Composition, ossia viene prodotta l'UI eseguendo per la prima volta le varie funzioni composable necessarie. In seguito, a causa del cambiamento dello stato di uno o più elementi della UI viene avviata la Recomposition: Compose esegue nuovamente tutte le funzioni che possono aver subito un cambiamento e in questo modo aggiorna la UI. L'interfaccia utente, infatti, è considerata immutabile e l'unico modo per modificarla è generarla nuovamente tenendo conto dei cambiamenti da effettuare.
È importante notare che durante questo processo Compose non riesegue tutte le funzioni coinvolte nella Composition, ma solo quelle i cui parametri sono mutati o che utilizzano al loro interno funzioni composable sensibili al dato cambiamento di stato. Questa feature, chiamata Intelligent Recomposition, permette di risparmiare tempo, potenza computazionale e batteria.
Per facilitare il compilatore durante la gestione della Recomposition e ottimizzare la procedura, sono stati introdotti i cosiddetti tipi stabili. Un tipo di dato stabile è marcato dall'annotazione @Stable e deve rispettare le seguenti regole:

- date due istanze, il risultato di equals è sempre lo stesso. Implicitamente dunque, ogni cambiamento di una delle due istanze deve essere riflesso dall'altra
- tutti gli attributi public sono stabili
- quando un attributo public viene modificato, Compose viene notificato del cambiamento

Tutti i tipi primitivi, le stringhe e in generale tutti i tipi immutabili sono considerati stabili. Applicando questo concetto ad una funzione, si parla di stabilità solo nel caso in cui la funzioni ritorni lo stesso risultato se sono passati gli stessi parametri, sottintendendo che tutti i parametri siano di un tipo stabile.

Quando viene avviata la Recomposition, i valori dei parametri sono confrontati per eguaglianza e se non sono cambiati la funzione viene saltata.

4.3 Funzioni Composable e Recomposition

A causa del funzionamento stesso del meccanismo della Recomposition, al fine di evitare bug e malfunzionamenti, risulta necessario adottare determinati comportamenti all'interno del codice che implementa le funzioni composable.
Innanzitutto, le funzioni composable devono evitare di modificare variabili globali e di causare side-effect durante la loro esecuzione. Queste funzioni, infatti, a causa delle ottimizzazioni che il plug-in di JetPack Compose per il compilatore introduce, possono essere eseguite in qualsiasi ordine (anche se, ad esempio, dichiarate in un certo ordine all'interno di uno stesso scope) e addirittura in parallelo.
Inoltre, dato che una singola funzione può essere ricomposta un altissimo numero di volte, è necessario che le funzioni composable siano leggere e facilmente eseguibili. Nel caso sia necessario compiere operazioni complesse, come ad esempio il caricamento di alcuni dati da un database o l'utilizzo di un'animazione, si devono utilizzare strumenti dedicati come, ad esempio, le coroutine, al fine di eseguire queste operazioni in background e non bloccare il main thread.

4.4 Gestione dello stato

In Jetpack Compose, lo stato è memorizzato in variabili di tipo State<T> e, in particolare, MutableState<T>, sua sottoclasse. Queste variabili detengono un solo attributo, value, di tipo T, che rappresenta il valore dello stato. Come illustrato in precedenza, il cambiamento di questo valore causa la Recomposition di tutte le sezioni della Composition che fanno riferimento ad esso.
Lo scopo del tipo State è dunque quello di fare da contenitore per il valore dello stato e di renderlo observable all'interno dell'applicazione, ossia causa scatenante della Recomposition.

4.5 Instance state

In JetPack Compose, l'instance state è strettamente legato al ciclo di vita delle funzioni composable, costituito da tre fasi:

- ingresso nella Composition
- Recomposition (zero o più volte)
- uscita dalla Composition

Il toolkit fornisce delle funzioni composable apposite, utilizzabili dallo sviluppatore in base alle necessità dell'applicazione:

- **remember {mutableStateOf(initialValue: T)}**

  Utilizzando queste due funzioni in maniera combinata, è possibile salvare un determinato instance state. Tuttavia, quando la funzione composable termina il proprio ciclo di vita e lascia la Compostion, lo stato è reinizializzato al suo valore di default.
  La funzione remember() è utilizzata per preservare il valore passatogli come parametro attraverso le varie Recomposition, in modo che non venga reinizializzato ogni qual volta lo scope a cui appartiene viene rieseguito. mutableStateOf(initialValue: T) invece si occupa di ritornare un oggetto di tipo MutableState<T> il cui attributo value è settato a initalValue.

- R**ememberSaveable {mutableStateOf(initialValue: T)}**

  Esattamente come visto al punto precedente, anche l'utilizzo di queste due funzioni ritorna un oggetto di tipo MutableState<T>. Tuttavia, a differenza di remember(), rememberSaveable() permette al valore passatogli come parametro di sopravvivere non solo alle Recomposition, ma anche alla ricreazione dell'activity. Con questo non si intende che lo stato viene salvato in maniera persistente, ma che è salvato e riutilizzato in quelle occasioni in cui l'applicazione viene distrutta ma l'utente non ne è a conoscenza. È il caso, ad esempio, di un'applicazione lasciata in background e non volontariamente terminata dall'utente: il sistema operativo potrebbe distruggerla per ottenere le risorse utilizzate dall'app, ma l'utente si aspetta di trovare l'applicazione nello stato in cui l'aveva lasciata prima di mandarla in background.

4.6 Persistent state

Per quanto riguarda il persistent state, JetPack Compose non ha introdotto alcun nuovo meccanismo o funzione. È dunque necessario utilizzare SharedPreferences.

4.7 State hoisting e Unidirectional Data Flow

Quando lo stato di una funzione composable rappresenta dettagli implementativi relativi alla funzione stessa, non modificabili e non riutilizzabili all'esterno di essa, è possibile memorizzare al suo interno il dato stato.
Tuttavia, è preferibile gestire lo stato il più in alto possibile all'interno della gerarchia delle funzioni e creare così funzioni "stateless composable", ossia funzioni controllate dal chiamante, che non gestiscono autonomamente il proprio stato, ma lo ricevono dall'esterno come parametro.
Questo paradigma di programmazione prende il nome di State Hoisting ed adottandolo si riescono a creare funzioni composable riutilizzabili in contesti diversi,

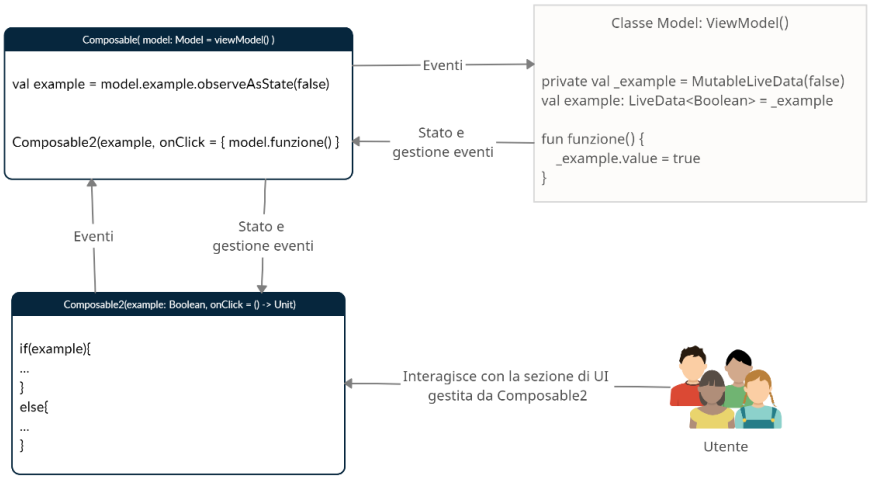
in quanto il loro stato viene controllato dalla funzione che le invoca, facilmente testabili e costituite da codice completamente slegato dal modo in cui lo stato viene memorizzato. Attraverso l'implementazione dello State Hoisting risulta naturale anche l'implementazione di un altro pattern, chiamato Unidirectional Data Flow.
Il cambiamento dello stato è, infatti, causato dagli eventi: l'utente clicca su un bottone, seleziona una checkbox, apre un menù ... Come illustrato precedentemente, attraverso l'hoisting lo stato, quando necessario, viene gestito più in alto nella gerarchia rispetto alla funzione composable che gestisce il singolo elemento della UI. Allo stesso modo, dunque, vengono gestiti gli eventi, attraverso una serie di callback che hanno origine nella funzione composable che si occupa di gestirle secondo le metodologie definite dall'Hoisting. In questo modo dunque, lo stato scorre "verso il basso", mentre gli eventi vanno "verso l'alto". Immaginando, ad esempio, un bottone che viene premuto, il flusso di esecuzione causa la gestione dell'evento e l'attivazione della callback invocata sullo onClick() del bottone e, di conseguenza, il cambiamento dello stato e l'attivazione della Recomposition.

4.8 ViewModel e Compose

La classe ViewModel è ideata al fine di memorizzare e gestire i dati relativi alla UI in maniera coerente rispetto al ciclo di vita dell'applicazione. Il suo scopo è dunque quello di estrarre la logica dell'interfaccia utente dalle classi che si occupano di istanziarla, rendendo così il codice più modulare e facilmente testabile. Inoltre, le istanze di ViewModel sono in grado di sopravvivere ai cambiamenti di configurazione, risolvendo così il problema del salvataggio dei dati e ottimizzando il loro utilizzo, evitando la ripetizione di eventuali operazioni temporalmente onerose da eseguire in maniera asincrona ogni volta che l'activity viene ricreata.

Risulta quindi naturale l'utilizzo di questa classe in applicazioni che utilizzano Jetpack Compose, in quanto permette di applicare nuovamente i principi dello State Hoisting e dello Unidirectional Data Flow.
Lo stato dei vari elementi della UI viene infatti ulteriormente estratto e memorizzato grazie a variabili di tipo LiveData, classe ideata per contenere i singoli data field degli oggetti ViewModel. Allo stesso modo, nel ViewModel vengono definite le varie funzioni necessarie a gestire gli eventi.

Nell'esempio sottostante viene mostrato questo processo. Particolare attenzione va riposta nei confronti del metodo observeAsState(initialValue: T) che permette di recuperare il valore slavato all'interno di un oggetto di tipo LiveData e di trasformarlo in una variabile di tipo MutableState<T>.


Composable(model: Model = viewModel())
val example = model.example.observeAsState(false)
Composable2(example, onClick = { model.funzione() }
Eventi
Stato e
gestione eventi
Classe Model: ViewModel()
private val _example = MutableLiveData(false)
val example: LiveData<Boolean> = _example
fun funzione() {
_example.value = true
}
Eventi
Stato e
gestione eventi
Composable2(example: Boolean, onClick = () -> Unit)
if(example){
...
}
else{
...
}
Interagisce con la sezione di UI
gestita da Composable2
Utente

5 Coroutine

Con coroutine si intende un design pattern concorrente, introdotto a partire dalla versione 1.3 di kotlin, utilizzato in android al fine di semplificare il codice che viene eseguito in modo asincrono.
Esse aiutano la gestione e l'esecuzione di operazioni che richiedono una notevole quantità di tempo per essere completate. A differenza dei thread, non vengono gestite dal sistema operativo, ma dal componente Kotlin Runtime. Inoltre, non sono legate ad un particolare thread, ovvero possono essere eseguite su un thread, sospese, e successivamente processate su un thread diverso.
Per poter utilizzare le coroutine all'interno del proprio progetto vi è la necessità di aggiungere la seguente dipendenza nel file build.gradle

```
dependencies {
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-android:1.3.9")
}
```

Tutte le coroutine devono essere eseguite all'interno di uno scope appropriato. Un coroutineScope si occupa di gestire una o più coroutine correlate.
Analogamente a quanto avviene con i thread le coroutine possono essere eseguite in diversi processi al fine di alleggerire il carico di lavoro da parte del MainThread.
Per avviare una coroutine vi è il bisogno di utilizzare l'istruzione launch, che si occupa di far eseguire in maniera concorrente la coroutine, in modo indipendente dal flusso di esecuzione principale. Una nuova coroutine può solamente essere eseguita all'interno di uno specifico CoroutineScope, il quale ne delimita la durata. I tre principali scope sono:

- **GlobalScope**: le coroutine avviate all'interno di questo scope "sopravvivono" per l'intero ciclo di vita di un'applicazione.
- **LifeCycleScope**: è simile al precedente, con la differenza che tutte le coroutine lanciate all'interno di un activity "muoiono" alla chiusura dell'activity.
- **ViewModelScope**: il ciclo di vita della coroutine è lo stesso del view model. All'interno della coroutine si possono richiamare le funzioni dichiarate con la parola chiave *suspend*. Queste particolari funzioni implementano caratteristiche aggiuntive rispetto alle funzioni standard. Esse possono essere possono essere messe in pausa, e poi riprese in un secondo momento. Quindi, Permettono di eseguire operazioni onerose senza dover bloccare il principale flusso di esecuzione del codice.

Esistono 5 tipi di Diaspatchers, che vengono passati al launcher delle coroutine come flag per indicare in quale tipo di processo si desidera che la coroutine venga eseguita.

- **Dispatchers.Main**: è il main thread.

- **Dispatchers.IO**: utilizzato per i processi che riguardano le interazioni con la rete e la memoria
- **Dispatchers.Default**: per indicare qualsiasi altro thread che non sia il Main, viene assegnato automaticamente.
- **Dispatchers.Unconfined**: si tratta di un flag particolare che prevede il cambiamento del processo sul quale una determinata azione viene effettuata quando essa viene sospesa e riattivata.
- **newSingleThreadContext**: permette all'utente di definire il proprio processo di esecuzione in modo manuale. Richiede di essere chiuso automaticamente al termine delle task per le quali è stato creato

Nel caso in cui non venisse fornito nessun dispatchers la coroutine verrà eseguita nello stesso processo della funzione che l'ha invocata.

6 Design

6.1 Modifier

Attraverso l'interfaccia Modifier è possibile modificare determinati aspetti relativi al modo in cui viene mostrato un elemento della UI. Per mezzo delle varie sottoclassi fornite da JetPack Compose, è infatti possibile aggiungere decorazioni, come il colore di background o la dimensione dell'elemento, e funzionalità, come ad esempio "clickable" o "zoomable".
Si comportano dunque in maniera simile a quella dei layout View-based, ma senza ovviamente essere inseriti in file XML.
In particolare, è possibile utilizzare i vari Modifier in maniera concatenata e verranno dunque applicati nell'ordine in cui sono dichiarati. Di conseguenza, un diverso ordine potrebbe causare un risultato diverso.

6.2 Theming and Material Design

JetPack Compose implementa il Material Design attraverso la libreria android.compose.material. Oltre a migliorare il design dei componenti dell'app, come bottoni, card, surface . . . , Compose implementa anche il Material Theming, che aiuta a rendere la grafica più consistente rendendo l'app nell'insieme più originale e personalizzata.
Per poterlo utilizzare, è necessario aggiungere la dipendenza nel file build.gradle dell'applicazione:

```
implementation("androidx.compose.material:material:1.0.0-beta08")
```

6.3 Color

Con il Material Design di JetPack Compose è possibile creare palette personalizzate da usare tra i vari componenti dell'app in maniera facile e ordinata. I colori sono definiti nel file Color.kt, che contiene anche informazioni utili per la darkTheme, in quanto è possibile dichiararvi i lightColors e i darkColors, set di colori appartenenti alla data categoria.

Tutti i componenti Material sono configurati per avere un colore di default dichiarato in una delle seguenti tipologie:

- primary o secondary
- varianti di primary o secondary

- colori specifici per background, surface, error, typography...
- “on” colors

Questi ultimi sono associati ai colori delle Surface in quanto indicano i colori dei componenti che vi appaiono di fronte, come testi o icone, che senza specificazione userebbero il proprio primary o secondary color.

```
val Gray800 = Color( color: 0xFF424242)
val Gray900 = Color( color: 0xFF212121)
val DeepPurple500 = Color( color: 0xFF673AB7)
val DeepPurple700 = Color( color: 0xFF512DA8)
val DeepPurple100 = Color( color: 0xFFD1C4E9)
val Complementary = Color( color: 0xFF89B73A)
val Red500 = Color( color: 0xFFD32F2F)
val Red700 = Color( color: 0xFFB71C1C)
val DeepOrange100 = Color( color: 0xFFFFECB3)
val Amber600 = Color( color: 0xFFFFA000)


val LightColors = lightColors(
    primary = Red500,
    primaryVariant = Red700,
    secondary = Amber600,
    surface = DeepOrange100
)

val DarkColors = darkColors(
    primary = Gray800,
    primaryVariant = Gray900,
    secondary = Amber600,
    onPrimary = Color.White,
    onSecondary = Color.Black
)
```

6.4 Typography

In modo simile a Color, Material permette di definire i vari stili di tipografia da utilizzare nell’applicazione nel file Type.kt
In questo file è possibile dichiarare possibili stili tipografici da richiamare in seguito su qualsiasi componente che utilizzi del testo.

```
val Typography = Typography(
    body1 = TextStyle(
        fontFamily = FontFamily.Default,
        fontWeight = FontWeight.Normal,
        fontSize = 16.sp
    ),

    button = TextStyle(
        fontFamily = FontFamily.Default,
        fontWeight = FontWeight.W500,
        fontSize = 14.sp
    ),

    caption = TextStyle(
        fontFamily = FontFamily.Default,
        fontWeight = FontWeight.Normal,
        fontSize = 12.sp
    )
)
```

6.5 Implementazione del tema e DarkTheme

Per implementare effettivamente il tema scelto all'interno dell'applicazione, si deve utilizzare la funziona composable MaterialTheme() che accetta come parametri i Colors, Typography e Shapes definiti in precedenza.
Questa funzione dev'essere chiamata prima di qualsiasi componente Material della UI in modo che ad esso venga applicato il tema adeguato.
Per quanto riguarda la DarkTheme, Compose Material mette a disposizione la funzione isSystemInDarkTheme che ritorna un boolean a seconda che il dispositivo su cui è installata l'applicazione abbia attiva la DarkTheme o meno o, in alternativa a seconda delle esigenze dell'applicazione, è possibile gestire la cosa considerando il temo come uno stato dell'applicazione.
in qualsiasi caso, risulta semplice implementare il MaterialTheme con i colori dell'interfaccia adeguati utilizzando le funzioni lightColors e darkColors dichiarate in Color.kt.
Nell'esempio sottostante, una possibile implementazione:

```
@Composable
fun MyApplicationTheme(
    darkTheme: MutableState<Boolean>,
    content: @Composable () -> Unit
) {
    val colors = if (darkTheme.value) {
        DarkColors
    } else {
        LightColors
    }

    MaterialTheme(
        colors = colors,
        typography = Typography,
        shapes = Shapes,
        content = content
    )
}
```

6.6 Shape

Material definisce solamente tre categorie di forme da poter definire in Shape.kt: small, medium e large. Per ognuna di esse è possibile personalizzare la forma, il tipo di angoli e la grandezza.

```
val Shapes = Shapes(
    small = RoundedCornerShape(4.dp),
    medium = RoundedCornerShape(4.dp),
    large = RoundedCornerShape(0.dp)
)
```

6.7 Orientazione e dimensione dello schermo

Come illustrato in precedenza, ogni qual volta lo stato di uno o più elementi della UI cambia, viene avviata la Recomposition e vengono rigenerati i componenti suscettibili al dato cambiamento.
Il toolkit dunque, sfruttando la sua integrazione con Kotlin, sopperisce all'assenza dei file xml dedicati ai layout attraverso questo meccanismo, effettuando un controllo a runtime di orientazione e dimensioni dello schermo. È possibile infatti ottenere queste informazioni attraverso funzioni come LocalConfiguration, per l'orientazione, Configuration.screenWidthDp e Configuration.screenHeightDp, per la larghezza e l'altezza.
Nell'esempio sottostante, viene effettuato un controllo dell'orientazione corrente dello schermo "statico", incapace di rilevare un cambio di configurazione dinamico, ed in base al valore ottenuto si decide quale UI mostrare.

```
val configuration = LocalConfiguration.current

when(configuration.orientation) {
    Configuration.ORIENTATION_LANDSCAPE -> {
        // Invoco funzione composable che descrive la UI
        // in configurazione landscape
    }
    else -> {
        // Invoco funzione composable che descrive la UI
        // in configurazione portrait
    }
}
```

6.8 Caso d'uso: RecyclerView

Utilizzando una programmazione View-based, per implementare un RecyclerView è necessario implementare il RecyclerViewAdapter, il rispettivo ViewHolder, tutte le funzioni ad essi relative ed inserire il file XML relativo al layout del singolo elemento della lista.
Jetpack Compose riesce a realizzare tutto questo utilizzando meno codice e mantenendo prestazione ottimali attraverso l'interfaccia LazyListScope. Questa interfaccia viene utilizzata sia da LazyColumn che da LazyRow per l'implementazione di liste scrollabili verticali e orizzontali.
Nell'esempio sottostante è realizzata una funzione composable, chiamata RecyclerWithCompose, che implementa le funzionalità di un RecyclerView in sole venti righe di codice.
Scopo di questa funzione è dunque quello di realizzare una lista scrollabile verticalmente. Gli elementi della lista sono tutti i numeri naturali compresi tra 1 e 50: in caso di numero pari, la riga della lista ha colore di sfondo giallo; in caso di numero dispari, la riga ha colore di sfondo ciano. I colori sono utilizzati attraverso la classe Color.kt e sono applicati attraverso uno specifico Modifier.

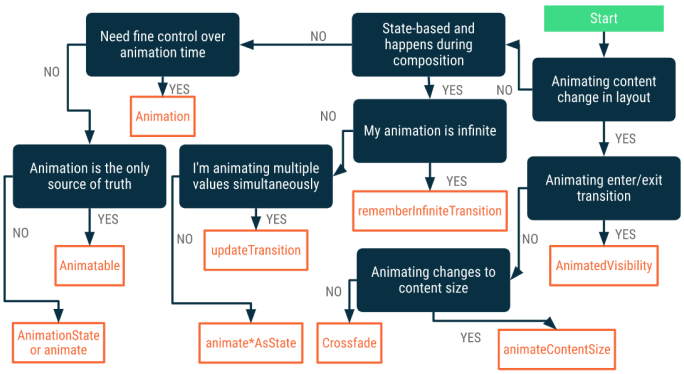
```
@Composable
fun RecyclerWithCompose() {
    val numeri = Array( size: 50) {it + 1}

    LazyColumn{  this: LazyListScope
        items(numeri) { numero ->
            Row(
                modifier = Modifier
                    .fillParentMaxWidth()
                    .background(
                        if(numero % 2 == 0)
                            Color.Cyan
                        else
                            Color.Yellow
                    )
            ){  this: RowScope
                Text(numero.toString())
            }
        }
    }
}
```

7 Animation

Jetpack Compose mette a disposizione API per la gestione di diversi tipi di animazioni per l'interfaccia utente dell'applicazione.
Le API sono classificate in due categorie:

- API a basso livello
- API ad alto livello, costruite utilizzando come base quelle a basso livello per rendere disponibili animazioni comuni e di facile approccio

Per avere più possibilità di personalizzazione è possibile inoltre appoggiarsi all'API Animation, alla quale si appoggiano a loro volte anche le API a basso livello.



Attraverso questo diagramma di flusso messo a disposizione dalla documentazione ufficiale, è possibile decidere facilmente quale tra le varie API è meglio usare per implementare l'animazione scelta.
Le API fornite sono le seguenti:

- AnimatedVisibility (experimental)
- Modifier.animateContentSize
- Crossfade
- rememberInfiniteTransition
- updateTransition
- animate*AsState
- Animation
- Animatable

- AnimationState or animate

E' buona norma ricordare che alcune di esse, essendo Experimental API, potrebbero essere rimosse o modificate in update futuri di Compose, e per questo han bisogno della notazione @ExperimentalAnimationApi per poter funzionare a dovere e ottimizzare la compilazione.

7.1 Preview

Grazie all'utilizzo della notazione @Preview che Compose offre, è possibile ispezionare le varie animazioni senza dover utilizzare un emulatore.
Android studio, tramite Compose Preview, permette la visualizzazione frame by frame, l'ispezione dei valori delle transizioni nell'animazione e l'animazione stessa

8 Navigation

Per rispondere alla necessità degli utenti di muoversi all'interno delle diverse content area che le applicazioni forniscono, JetPack Compose fornisce uno strumento chiamato Navigation, ancora in corso di sviluppo e attualmente utilizzabile nella sua versione alfa.
Basato sulle caratteristiche e infrastrutture introdotte dal Navigation Component View-based, la versione Compose si propone di sfruttare al massimo le opportunità aperte dal toolkit, portando dunque lo sviluppatore a costruire applicazioni caratterizzate da una singola activity, che non utilizzano intent e fragment a questo scopo e che, di conseguenza, si affidano solamente alle funzioni composable per la gestione dell'interfaccia utente.
Per poterlo utilizzare, è necessario aggiungere la dipendenza nel file build.gradle dell'applicazione:

```
implementation("androidx.navigation:navigation-compose:2.4.0-alpha01")
```

8.1 Principi

Navigation si basa sul concetto di destination.
Una destination è una delle differenti content area dell'applicazione, uno degli schermi visualizzabili dall'utente, un qualunque "luogo" raggiungibile dall'utente spostandosi all'interno dell'applicazione stessa.
Ogni applicazione ha una start destination fissata: si tratta del primo schermo che l'utente vede quando l'app viene eseguita e dell'ultimo schermo che l'utente incontra prima di tornare al launcher a forza di premere il pulsante back. Quando l'app viene eseguita, dunque, viene mostrata la start destination. Questa diventa la destination base del back stack, mentre in cima (immaginando lo stack come una pila) si trova lo schermo visualizzato in quel momento. In mezzo, sono accatastate le destination attraversate dall'utente per arrivare a quella attualmente visualizzata.
Per eseguire il pop del back stack e tornare alla destination precedentemente visualizzata, l'utente può servirsi dei tasti Back e Up (la freccia che compare in alto sinistra).
La Navigation avviene dunque tra le varie destination dell'app.
Un Navigation Graph, costrutto ereditato dal Navigation Component, contiene tutte le destination dell'app. Ogni activity ha il proprio navigation graph.

8.2 Utilizzo

A livello di codice, gli elementi principali sono due:

- **navController**
  Reperibile invocando la funzione rememberNavController(), tiene traccia del back stack e dello stato di ogni schermo. In maniera simile a quanto visto con lo State Hoisting, dovrebbe essere istanziato, all'interno della gerarchia delle funzioni composable, nel luogo in cui tutte le funzioni che devono utilizzarlo possono accedere ad esso.

- **NavHost**
  Si tratta di una funzione composable che ha il compito di creare il Navigation Graph e di associarlo al navController precedentemente istanziato. Ogni navController è legato ad un solo Navigation Graph, dunque interagisce con un solo NavHost. Per creare il Navigation Graph all'interno di NavHost, attraverso la funzione composable(), ogni destination è associata ad una route: una stringa che rappresenta univocamente il "percorso" necessario per raggiungere la data destination.

Sintassi:

```
NavHost(navController, startDestination = "route1") {  this: NavGraphBuilder
    composable( route: "route1") { Destination1() }
    composable( route: "route2") { Destination2() }
    composable( route: "route3") { Destination3() }

}
```

Per effettuare la navigazione e passare da una determinata destination ad un'altra, si utilizza l'istanza di navController e la funzione navigate(), che accetta come parametro una stringa corrispondente ad una delle route inserite nel NavHost a cui il navController è associato. Dal momento che il controller tiene traccia del back stack, utilizzando la sintassi delle Higher-Order function di Kotlin, è possibile utilizzare una lambda per manipolarlo attraverso l'invocazione di funzioni come popUpTo().
Inoltre, JetPack Compose fornisce una funzione composable, chiamata BackHandler, che permette di sovrascrivere il comportamento del pulsante back.

9 Interoperabilità

Il framework Jetpack Compose è stato realizzato in modo tale da risultare compatibile anche con gli altri toolkit già presenti per la realizzazione dell'interfaccia utente.
Le opzioni che permettono di far cooperare jetpack Compose con Android View sono:

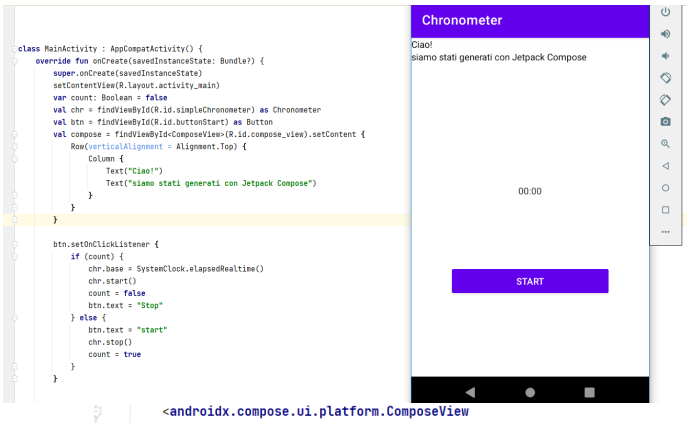
- **ComposeView**
- **AndroidView**

Compose inoltre, mette anche a disposizione librerie per accedere alle risorse presenti nel progetto Android.
Di seguito sono riportate le funzioni per accedere alle risorse più comuni:

- Tramite la funzione stringResource(R.string.id) si può ottenere la stringa con il dato id, definita nel file string.xml;
- Con colorResource(R.color.name) per ottenere il valore del colore avente come nome "name";
- Si utilizza painterResource(id=R.drawable.immagine) per caricare all'interno del proprio codice Kotlin una data immagine.

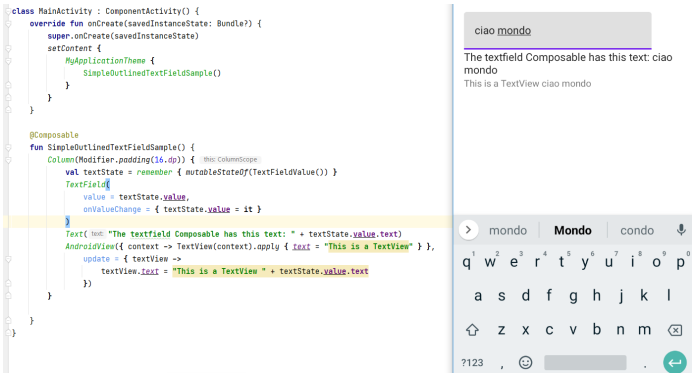
9.1 Compose View

Questa modalità di interoperabilità è da adottare quando si ha la necessità di aggiungere funzionalità realizzate in Jetpack Compose ad un'applicazione esistente, realizzata con Android View.
All'interno del file XML che contiene il layout della UI si aggiunge un attributo di nome ComposeView. Successivamente dal codice Kotlin, nel quale vogliamo inserire le nuove funzionalità create con Compose, si ottiene il riferimento a tale oggetto tramite il metodo findViewById. Una volta ottenuta la reference di ComposeView, si invoca il metodo setContent() su di essa, e all'interno si inserisce il codice Compose.
Di seguito viene riportato un esempio di codice. Nell'esempio in questione, viene aggiunta una semplice Label ad un'applicazione che esemplifica il funzionamento di un cronometro, realizzato con android View.



9.2 Android View in Compose

AndroidView permette di eseguire l'operazione inversa, ovvero di integrare alla parte di codice scritta secondo il Framework Jetpack Compose layout realizzati con l'approccio View.
Questo tipo di integrazione permette di utilizzare le tradizionali componenti grafiche descritte da View, all'interno delle funzioni @Composable. È essenziale utilizzare questo formalismo per implementare componenti grafici non ancora disponibili in Jetpack Compose, come ad esempio AdView o MapView.
AndroidView accetta anche come parametro una funzione lambda di nome update, che viene invocata ogni qualvolta ci sia la necessità di effettuare la recomposition della view.
Nell'esempio riportato in seguito, si è realizzato una semplice applicazione che permette all'utente di inserire del testo. Esso verrà a sua volta mostrato a runtime sia dalla funzione composable Text, che dalla TextView, realizzata tramite AndroidView.



10 Compose Desktop

Al momento JetBrains sta procedendo con un ulteriore sviluppo del framework, con lo scopo di realizzare applicazioni Cross-Platform. Questa particolare versione, denominata Compose for Desktop attualmente si presenta nella versione Alpha. Permette di progettare Applicazioni per macOs, Linux, Windows, basate sugli stessi principi della versione android, ovvero scritte in Kotlin utilizzando un approccio dichiarativo per la realizzazione dell'interfaccia utente. Inoltre, permette di utilizzare le API di molti framework attuali, come React e Flutter.
Questo progetto si basa su JetPack Compose, e quindi di conseguenza le conoscenze apprese per lo sviluppo con uno dei due framework si possono applicare direttamente all'altro. Infatti, Compose for Desktop viene sviluppato con il supporto di Google in modo tale da assicurare uno sviluppo congiunto delle due tecnologie.
Uno dei punti di forza di essa riguarda la possibilità di condividere parte dell'implementazione dell'interfaccia utente tra la versione desktop e mobile.
Per la completa integrazione con il mondo desktop il framework viene integrato con API specifiche per le funzionalità desktop. Queste API ad esempio permettono di riconoscere il click del mouse, il ridimensionamento della finestra, e altro ancora.
Per iniziare a sviluppare queste applicazioni JetBrains mette a disposizione un plugin integrabile con Intellij. Inoltre, la minima JDK richiesta per questo tipo di applicazioni è la versione 11.

11 La nostra Applicazione in Compose

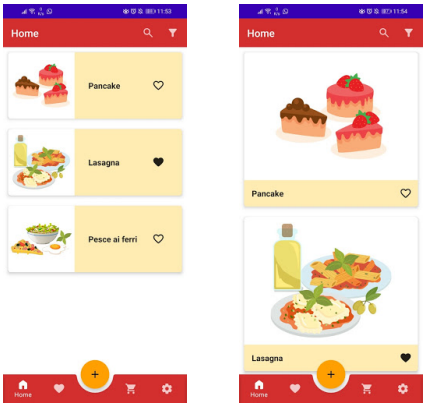
Dopo un'attenta analisi di Compose, l'applicazione da noi implementata che ne esemplifica le varie caratteristiche utilizza l'approccio dichiarativo tipico di Jetpack Compose, interlacciando le varie funzionalità tramite l'interoperabilità con AndroidView, ViewModel e un utilizzo pratico di SQL Lite.
Nei paragrafi seguenti è stato scelto di mostrare in breve le funzioni principali dell'app per permetterne un utilizzo più veloce ed efficace.
L'applicazione sviluppata si pone l'obiettivo di diventare un ricettario digitale per tenere in memoria le ricette dell'utente, permettendone la preview e la modifica in caso di necessità. Ad ogni ricetta verrà associato un titolo di riconoscimento, una descrizione, una o più tipologie (es. Primo piatto, Vegetariano ...), gli ingredienti e facoltativamente un'immagine di preview che l'utente può scegliere dalla galleria.

11.1 Preview

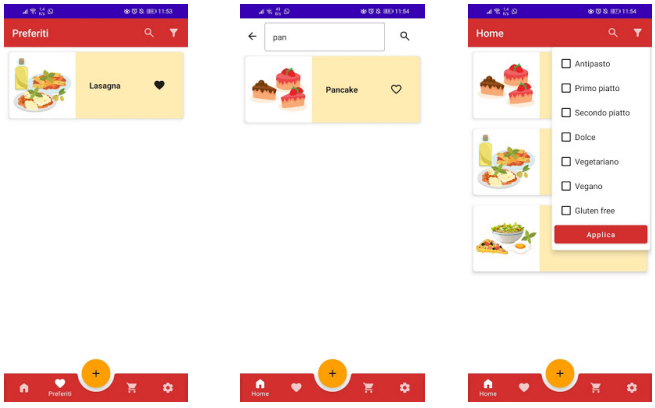
L'interfaccia principale dell'applicazione consiste nell'utilizzo di una BottomBar per navigare tra Home, Preferiti, Carrello e Modalità, e di un Pulsante che permetterà l'aggiunta delle ricette al proprio ricettario. Per ogni ricetta esiste una pagina in dettaglio che ne mostra ingredienti, filtri, descrizione e l'immagine che la rappresenta e potrà essere eliminata o modificata a scelta dell'utente.

11.2 Home e Preferiti

La cosiddetta 'Home' dell'applicazione è una semplice LazyColumn che contiene tutti i titoli delle ricette inserite dall'utente. Al primo accesso all'applicazione, la Home conterrà anche delle ricette di default come dimostrazione per l'interfaccia UI finale.



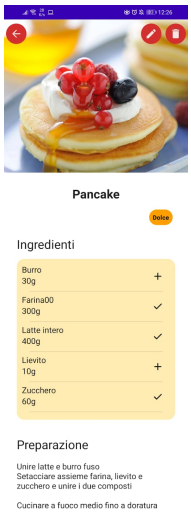
Si nota immediatamente la TopAppBar contenente due pulsanti che permettono la ricerca tramite Categoria e tramite l'inserimento di parole chiavi contenute nel titolo.
Ogni ricetta della lista viene visualizzata insieme a un'icona a cuoricino, che una volta cliccato salverà la ricetta selezionata come Preferito, e potrà essere visualizzata della tab Preferiti cliccando sulla relativa icona nella BottomBar Navigation.



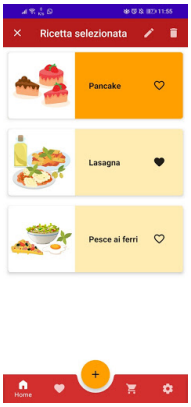
Ogni ricetta può essere cliccata in due modalità differenti:

- **onTap**: al click dell'utente, si aprirà la pagina Detail della ricetta selezionata, dove è possibile leggere in dettaglio tutte le informazioni disponibili e effettuare delle operazioni specifiche come l'eliminazione della ricetta, la sua modifica o l'aggiunta dei suoi ingredienti al Carrello.

- **onLongPress** tenendo premuta una data ricetta, questa verrà concepita dall'applicazione come selezionata, e la TopAppBar mostrerà due nuovi pulsanti, l'icona del modifica (che permette la completa modifica della ricetta, ad eccezione del titolo) e l'icona del cestino (che permette l'eliminazione della ricetta dal database)



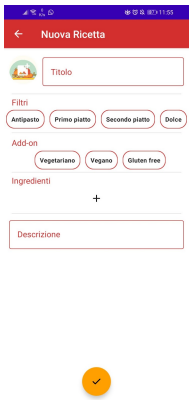
(a) *Dettaglio di una ricetta*



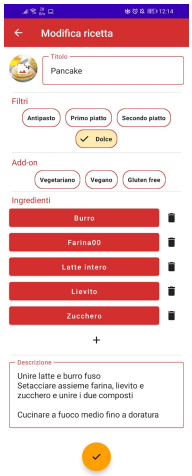
(b) *onLongPress*

11.3 Aggiungi e Modifica

Dopo aver premuto sul tasto Aggiungi già visibile nella Home, è possibile aggiungere una nuova ricetta all'applicazione.
E' piuttosto intuitivo come si possa aggiungere un titolo e una descrizione tramite le due TextField presenti nell'interfaccia e le categorie della ricetta selezionando le chip presenti nella sezione Filtri.
L'immagine invece sarà selezionabile cliccando sull'icona dell'immagine accanto al titolo, che aprirà l'interfaccia della galleria, mentre gli ingredienti si possono aggiungere premendo l'icona del + nella sezione Ingredienti. Ogni ingrediente inserito sarà modificabile cliccandoci sopra e eliminabile tramite l'icona del cestino relativa ad esso.
Non aver inserito alcuni elementi non è un problema, in quanto l'applicazione stessa ti ricorda i campi vuoti di cui ha bisogno per memorizzare correttamente la ricetta inserita.

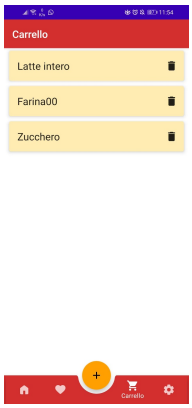


Alternativamente, ad un'interfaccia molto simile, è possibile arrivare tramite l'opzione Modifica, che si trova sia nella TopAppBar dopo il LongPress o nel dettaglio della ricetta da modificare.
Tramite questa interfaccia è possibile modificare ogni singolo aspetto della ricetta, fuorché il titolo.



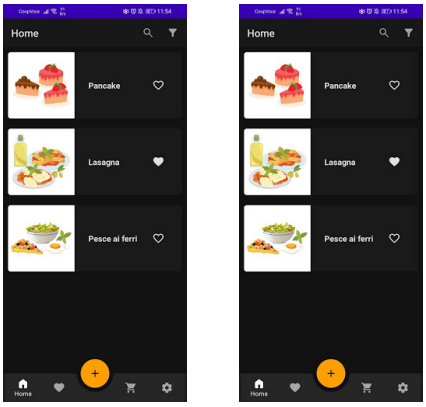
11.4 Carrello

Tramite la BottomBar Navigation è possibile visualizzare la facciata relativa al Carrello, contenente tutti gli ingredienti inseriti dall'utente. Ogni ingrediente può essere aggiunto al Carrello tramite le spunte relative agli ingredienti nelle pagine delle ricette in dettaglio, mentre dal Carrello stesso è possibile rimuovere gli ingredienti che non interessano più.



11.5 Modalità

L'ultima pagina della BottomBar si presenta come impostazioni di personalizzazione dell'applicazione.
Attraverso dei pulsanti è possibile passare dalla modalità Light alla modalità Dark, mantenendo la preferenza ai prossimi accessi all'app, oppure è possibile specificare la modalità di visualizzazione della lista delle ricette dando priorità all'immagine (GridView) o al testo (ListView)



12 Sitografia

12.1 Documentazione ufficiale

Build better apps faster withJetpack Compose
https://developer.android.com/jetpack/compose
Courses Jetpack Compose
https://developer.android.com/courses/pathways/compose
Compose
https://developer.android.com/jetpack/androidx/releases/compose
Jetpack Compose Samples
https://github.com/android/compose-samples
Jetpack Compose Codelabs
https://github.com/googlecodelabs/android-compose-codelabs
Canale youtube di AndroidDevelopers
https://www.youtube.com/c/AndroidDevelopers/search?query=Jetpack%20Compose
Jetpack Compose by Android Developers
https://www.youtube.com/watch?v=U5BwfqBpiWU&ab_channel=AndroidDevelopers
Tipi di dato Stable:
https://developer.android.com/reference/kotlin/androidx/compose/runtime/Stable
API Guidelines for Jetpack Compose
https://github.com/androidx/androidx/blob/androidx-main/compose/docs/compose-api-guidelines.md
Use Android Studio with Jetpack Compose
https://developer.android.com/jetpack/compose/setup
Compose for Desktop and Web, by JetBrains
https://github.com/JetBrains/compose-jb
Kotlin coroutines on Android
https://developer.android.com/kotlin/coroutines?gclid=EAIaIQobChMI8r3SpMaM8QIVTwOLCh11eQJ8EAA
BwE&gclsrc=aw.ds
Coroutines in Kotlin
https://developer.android.com/codelabs/kotlin-coroutines#3

12.2 Blog

What Is the Difference between Canary, Beta, RC and Stable Releases in Android Studio? by AndroidPub
https://medium.com/android-news/what-is-the-difference-between-canary-beta-rc-and-stable-releases-in-the-android-studio-bbbb77e7c3cf
https://medium.com/android-news/what-is-the-difference-between-canary-beta-rc-and-stable-releases-in-the-android-studio-bbbb77e7c3cf
https://proandroiddev.com/the-journey-of-jetpack-compose-i-eef660bc546b
What's new in Jetpack by Florina Muntenescu

https://medium.com/androiddevelopers/whats-new-in-jetpack-1891d205e136
Understanding Jetpack Compose — part 1 of 2
https://medium.com/androiddevelopers/understanding-jetpack-compose-part-1-of-2-ca316fe39050
Jetpack Compose: The Future of Android UI
https://link.medium.com/0gZ6DLIBReb
Jetpack Compose Tutorial for Android: Getting Started
https://www.raywenderlich.com/15361077-jetpack-compose-tutorial-for-android-getting-started
Under the hood of Jetpack Compose — part 2 of 2
https://medium.com/androiddevelopers/under-the-hood-of-jetpack-compose-part-2-of-2-37b2c20c6cdd
Cartographing Jetpack Compose: compiler and runtime
https://dev.to/tkuenneth/cartographing-jetpack-compose-compiler-and-runtime-1605
Jetpack Compose: What you need to know, pt. 1
https://www.mobileit.cz/Blog/Pages/compose-1.aspx
Jetpack Compose Interop Part 1: Using Traditional Views and Layouts in Compose with AndroidView
https://proandroiddev.com/jetpack-compose-interop-part-1-using-traditional-views-and-layouts-in-compose-with-androidview-b6f1b1c3eb1
Structured concurrency
https://elizarov.medium.com/structured-concurrency-722d765aa952
Jetpack Compose for Desktop: Milestone 1 Released
https://blog.jetbrains.com/cross-post/jetpack-compose-for-desktop-milestone-1-released/
Compose for Desktop
https://proandroiddev.com/compose-for-desktop-1cb05dff8710
Jetpack Compose... on the Desktop
https://commonsware.com/blog/2020/04/25/jetpack-compose-desktop.html
Android Jetpack Compose: Theme Made Easy
https://medium.com/mobile-app-development-publication/android-jetpack-compose-theme-made-easy-1812150239fe
How to Style and Theme an App With Jetpack Compose
https://www.freecodecamp.org/news/how-to-style-and-theme-an-app-with-jetpack-compose/
Coroutines basics
https://kotlinlang.org/docs/coroutines-basics.html#your-first-coroutine
Scopes in Kotlin Coroutines
https://www.geeksforgeeks.org/scopes-in-kotlin-coroutines/
Kotlin Coroutine Scope, Context, and Job made simple
https://medium.com/mobile-app-development-publication/kotlin-coroutine-scope-context-and-job-made-simple-5adf89fcfe94
Jetpack Compose: Now in Beta
https://material.io/blog/jetpack-compose-beta
Building a scroll-to-fade AppBar

https://compose.academy/blog/building_a_scroll-to-fade_topbar
Frequently Asked Questions
https://www.jetpackcompose.app/faq

12.3 GITHUB

Learn Jetpack Compose By Example
https://github.com/vinaygaba/Learn-Jetpack-Compose-By-Example#learn-jetpack-compose-by-example
SimpleComposeRowFlow
https://github.com/icesmith/android-samples/tree/master/SimpleComposeFlowRow
Hands on Kotlin
https://github.com/AseemWangoo/hands_on_kotlin