Side-effects in Compose

A **side-effect** is a change to the state of the app that happens outside the scope of a composable function. Due to composables' lifecycle and properties such as unpredictable recompositions, executing recompositions of composables in different orders, or recompositions that can be discarded, composables should ideally be side-effect free (/jetpack/compose/mental-model).

However, sometimes side-effects are necessary, for example, to trigger a one-off event such as showing a snackbar or navigate to another screen given a certain state condition. These actions should be called from a controlled environment that is aware of the lifecycle of the composable. In this page, you'll learn about the different side-effect APIs Jetpack Compose offers.

State and effect use cases

As covered in the Thinking in Compose (/jetpack/compose/mental-model) documentation, composables should be side-effect free. When you need to make changes to the state of the app (as described in the Managing state documentation (/jetpack/compose/state) doc), **you should use the Effect APIs so that those side effects are executed in a predictable manner**.

Key Term: An **effect** is a composable function that doesn't emit UI and causes side effects to run when a composition completes.

Due to the different possibilities effects open up in Compose, they can be easily overused. Make sure that the work you do in them is UI related and doesn't break **unidirectional data flow** as explained in the Managing state documentation (/jetpack/compose/state#unidirectional-data-flow-in-jetpack-compose).

Note: A responsive UI is inherently asynchronous, and Jetpack Compose solves this by embracing coroutines at the API level instead of using callbacks. To learn more about Coroutines, check out the Kotlin coroutines on Android (/kotlin/coroutines) guide.

LaunchedEffect: run suspend functions in the scope of a composable

To call suspend functions safely from inside a composable, use the `LaunchedEffect` (/reference/kotlin/androidx/compose/runtime/package-summary#LaunchedEffect(kotlin.Any,kotlin.coroutines.SuspendFunction1)) composable. When `LaunchedEffect` enters the Composition, it launches a coroutine with the block of code passed as a parameter. The coroutine will be cancelled if `LaunchedEffect` leaves the composition. If `LaunchedEffect` is recomposed with different keys (see the Restarting Effects (#restarting-effects) section below), the existing coroutine will be cancelled and the new suspend function will be launched in a new coroutine.

For example, showing a `Snackbar` (https://developer.android.com/reference/kotlin/androidx/compose/material3/package-summary#Snackbar(androidx.compose.material3.SnackbarData,androidx.compose.ui.Modifier,kotlin.Boolean,androidx.compose.ui.graphics.Shape,androidx.compose.ui.graphics.Color,androidx.compose.ui.graphics.Color,androidx.compose.ui.graphics.Color,androidx.compose.ui.graphics.Color,androidx.compose.ui.graphics.Color)) in a `Scaffold` (https://developer.android.com/reference/kotlin/androidx/compose/material3/package-summary#Scaffold(androidx.compose.ui.Modifier,kotlin.Function0,kotlin.Function0,kotlin.Function0,kotlin.Function0,androidx.compose.material3.FabPosition,androidx.compose.ui.graphics.Color,androidx.compose.ui.graphics.Color,androidx.compose.foundation.layout.WindowInsets,kotlin.Function1))) is done with the `SnackbarHostState.showSnackbar` (/reference/kotlin/androidx/compose/material3/SnackbarHostState) function, which is a suspend function.

```
@Composable
fun MyScreen(
    state: UiState<List<Movie>>,
    snackbarHostState: SnackbarHostState
) {

    // If the UI state contains an error, show snackbar
    if (state.hasError) {

        // `LaunchedEffect` will cancel and re-launch if
        // `scaffoldState.snackbarHostState` changes
        LaunchedEffect(snackbarHostState) {
            // Show snackbar using a coroutine, when the coroutine is cancell
            // snackbar will automatically dismiss. This coroutine will cance
            // `state.hasError` is false, and only start when `state.hasError
            // (due to the above if-check), or if `scaffoldState.snackbarHost
            snackbarHostState.showSnackbar(
                message = "Error message",
                actionLabel = "Retry message"
            )
        }
    }

    Scaffold(
        snackbarHost = {
            SnackbarHost(hostState = snackbarHostState)
        }
    ) { contentPadding ->
        // ...
    }
}
nippets/src/main/java/com/example/compose/snippets/sideeffects/SideEffectsSnippets.kt#L54-L95)
```

In the code above, a coroutine is triggered if the state contains an error and it'll be cancelled when it doesn't. As the `LaunchedEffect` call site is inside an if statement, when the statement is false, if `LaunchedEffect` was in the Composition, it'll be removed, and therefore, the coroutine will be cancelled.

rememberCoroutineScope: obtain a composition-aware scope to lau a coroutine outside a composable

As `LaunchedEffect` is a composable function, it can only be used inside other composable functions. In order to launch a coroutine outside of a composable, but scoped so that it will be automatically canceled once it leaves the composition, use `rememberCoroutineScope` (/reference/kotlin/androidx/compose/runtime/package-summary#rememberCoroutineScope(kotlin.Function0)) . Also use `rememberCoroutineScope` whenever you need to control the lifecycle of one or more coroutines manually, for example, cancelling an animation when a user event happens.

`rememberCoroutineScope` is a composable function that returns a `CoroutineScope` bound to the point of the Composition where it's called. The scope will be cancelled when the call leaves the Composition.

Following the previous example, you could use this code to show a `Snackbar` when the user taps on a `Button`:

```
@Composable
fun MoviesScreen(snackbarHostState: SnackbarHostState) {

    // Creates a CoroutineScope bound to the MoviesScreen's lifecycle
    val scope = rememberCoroutineScope()

    Scaffold(
        snackbarHost = {
            SnackbarHost(hostState = snackbarHostState)
        }
    ) { contentPadding ->
        Column(Modifier.padding(contentPadding)) {
            Button(
                onClick = {
                    // Create a new coroutine in the event handler to show a
                    scope.launch {
                        snackbarHostState.showSnackbar("Something happened!")
                    }
                }
            ) {
                Text("Press me")
            }
        }
    }
}
```
ippets/src/main/java/com/example/compose/snippets/sideeffects/SideEffectsSnippets.kt#L99-L123)

rememberUpdatedState: reference a value in an effect that shouldn't restart if the value changes

`LaunchedEffect` restarts when one of the key parameters changes. However, in some situations you might want to capture a value in your effect that, if it changes, you do not want the effect to restart. In order to do this, it is required to use `rememberUpdatedState` to create a reference to this value which can be captured and updated. This approach is helpful for effects that contain long-lived operations that may be expensive or prohibitive to recreate and restart.

For example, suppose your app has a `LandingScreen` that disappears after some time. Even if `LandingScreen` is recomposed, the effect that waits for some time and notifies that the time passed shouldn't be restarted:

```
@Composable
fun LandingScreen(onTimeout: () -> Unit) {

    // This will always refer to the latest onTimeout function that
    // LandingScreen was recomposed with
    val currentOnTimeout by rememberUpdatedState(onTimeout)

    // Create an effect that matches the lifecycle of LandingScreen.
    // If LandingScreen recomposes, the delay shouldn't start again.
    LaunchedEffect(true) {
        delay(SplashWaitTimeMillis)
        currentOnTimeout()
    }

    /* Landing screen content */
}
```
ppets/src/main/java/com/example/compose/snippets/sideeffects/SideEffectsSnippets.kt#L127-L145)

To create an effect that matches the lifecycle of the call site, a never-changing constant like `Unit` or `true` is passed as a parameter. In the code above, `LaunchedEffect(true)` is used. To make sure that the `onTimeout` lambda *always* contains the latest value that `LandingScreen` was recomposed with, `onTimeout` needs to be wrapped with the `rememberUpdatedState` function. The returned `State`, `currentOnTimeout` in the code, should be used in the effect.

Warning: `LaunchedEffect(true)` is as suspicious as a `while(true)`. Even though there are valid use cases for it, *always* pause and make sure that's what you need.

DisposableEffect: effects that require cleanup

For side effects that need to be *cleaned up* after the keys change or if the composable leaves the Composition, use `DisposableEffect` (/reference/kotlin/androidx/compose/runtime/package-summary#DisposableEffect(kotlin.Any,kotlin.Function1)) . If the `DisposableEffect` keys change, the composable needs to *dispose* (do the cleanup for) its current effect, and reset by calling the effect again.

As an example, you might want to send analytics events based on `Lifecycle` events (/topic/libraries/architecture/lifecycle#lc) by using a `LifecycleObserver`

(/reference/androidx/lifecycle/LifecycleObserver). To listen for those events in Compose, use a `DisposableEffect` to register and unregister the observer when needed.

```
@Composable
fun HomeScreen(
    lifecycleOwner: LifecycleOwner = LocalLifecycleOwner.current,
    onStart: () -> Unit, // Send the 'started' analytics event
    onStop: () -> Unit // Send the 'stopped' analytics event
) {
    // Safely update the current lambdas when a new one is provided
    val currentOnStart by rememberUpdatedState(onStart)
    val currentOnStop by rememberUpdatedState(onStop)

    // If `lifecycleOwner` changes, dispose and reset the effect
    DisposableEffect(lifecycleOwner) {
        // Create an observer that triggers our remembered callbacks
        // for sending analytics events
        val observer = LifecycleEventObserver { _, event ->
            if (event == Lifecycle.Event.ON_START) {
                currentOnStart()
            } else if (event == Lifecycle.Event.ON_STOP) {
                currentOnStop()
            }
        }

        // Add the observer to the lifecycle
        lifecycleOwner.lifecycle.addObserver(observer)

        // When the effect leaves the Composition, remove the observer
        onDispose {
            lifecycleOwner.lifecycle.removeObserver(observer)
        }
    }

    /* Home screen content */
}
```
ppets/src/main/java/com/example/compose/snippets/sideeffects/SideEffectsSnippets.kt#L149-L181)

In the code above, the effect will add the `observer` to the `lifecycleOwner`. If `lifecycleOwner` changes, the effect is disposed and restarted with the new `lifecycleOwner`.

A `DisposableEffect` must include an `onDispose` clause as the final statement in its block of code. Otherwise, the IDE displays a build-time error.

Note: Having an empty block in **`onDispose`** is not a good practice. Always reconsider to see if there's an effect that fits your use case better

SideEffect: publish Compose state to non-compose code

To share Compose state with objects not managed by compose, use the `SideEffect` (/reference/kotlin/androidx/compose/runtime/package-summary#SideEffect(kotlin.Function0)) composable, as it's invoked on every successful recomposition.

For example, your analytics library might allow you to segment your user population by attaching custom metadata ("user properties" in this example) to all subsequent analytics events. To communicate the user type of the current user to your analytics library, use `SideEffect` to update its value.

```
@Composable
fun rememberFirebaseAnalytics(user: User): FirebaseAnalytics {
    val analytics: FirebaseAnalytics = remember {
        FirebaseAnalytics()
    }

    // On every successful composition, update FirebaseAnalytics with
    // the userType from the current User, ensuring that future analytics
    // events have this metadata attached
    SideEffect {
        analytics.setUserProperty("userType", user.userType)
    }
    return analytics
}
```
ɔpets/src/main/java/com/example/compose/snippets/sideeffects/SideEffectsSnippets.kt#L185-L198)

produceState: convert non-Compose state into Compose state

`produceState` (/reference/kotlin/androidx/compose/runtime/package-summary#produceState(kotlin.Any,kotlin.coroutines.SuspendFunction1)) launches a coroutine scoped to the Composition that can push values into a returned `State` (/reference/kotlin/androidx/compose/runtime/State). Use it to convert non-Compose state into Compose state, for example bringing external subscription-driven state such as `Flow`, `LiveData`, or `RxJava` into the Composition.

The producer is launched when `produceState` enters the Composition, and will be cancelled when it leaves the Composition. The returned `State` conflates; setting the same value won't trigger a recomposition.

Even though `produceState` creates a coroutine, it can also be used to observe non-suspending sources of data. To remove the subscription to that source, use the `awaitDispose` (/reference/kotlin/androidx/compose/runtime/ProduceStateScope#awaitDispose(kotlin.Function0)) function.

The following example shows how to use `produceState` to load an image from the network. The `loadNetworkImage` composable function returns a `State` that can be used in other composables.

```
@Composable
fun loadNetworkImage(
    url: String,
    imageRepository: ImageRepository = ImageRepository()
): State<Result<Image>> {

    // Creates a State<T> with Result.Loading as initial value
    // If either `url` or `imageRepository` changes, the running producer
    // will cancel and will be re-launched with the new inputs.
    return produceState<Result<Image>>(initialValue = Result.Loading, url, im

        // In a coroutine, can make suspend calls
        val image = imageRepository.load(url)

        // Update State with either an Error or Success result.
        // This will trigger a recomposition where this State is read
        value = if (image == null) {
            Result.Error
        } else {
            Result.Success(image)
        }
    }
}
```
pets/src/main/java/com/example/compose/snippets/sideeffects/SideEffectsSnippets.kt#L202-L234)

Note: Composables with a return type should be named the way you'd name a normal Kotlin function, starting with a lowercase letter.

Key Point: Under the hood, **`produceState`** makes use of other effects! It holds a **`result`** variable using **`remember { mutableStateOf(initialValue) }`**, and triggers the **`producer`** block in a **`LaunchedEffect`**. Whenever **`value`** is updated in the **`producer`** block, the **`result`** state is updated to the new value.

You can easily create your own effects building on top of the existing APIs.

derivedStateOf: convert one or multiple state objects into another st

Use `derivedStateOf` (/reference/kotlin/androidx/compose/runtime/package-summary#derivedStateOf(kotlin.Function0)) when a certain state is calculated or derived from other state objects. Using this function guarantees that the calculation will only occur whenever one of the states used in the calculation changes.

The following example shows a basic *To Do* list whose tasks with user-defined high priority keywords appear first:

```
@Composable
fun TodoList(highPriorityKeywords: List<String> = listOf("Review", "Unblock",

    val todoTasks = remember { mutableStateListOf<String>() }

    // Calculate high priority tasks only when the todoTasks or highPriorityK
    // change, not on every recomposition
    val highPriorityTasks by remember(highPriorityKeywords) {
        derivedStateOf {
            todoTasks.filter { task ->
                highPriorityKeywords.any { keyword ->
                    task.contains(keyword)
                }
            }
        }
    }

    Box(Modifier.fillMaxSize()) {
        LazyColumn {
            items(highPriorityTasks) { /* ... */ }
            items(todoTasks) { /* ... */ }
        }
        /* Rest of the UI where users can add elements to the list */
    }
}
ppets/src/main/java/com/example/compose/snippets/sideeffects/SideEffectsSnippets.kt#L238-L262)
```

In the code above, `derivedStateOf` guarantees that whenever `todoTasks` changes, the `highPriorityTasks` calculation will occur and the UI will be updated accordingly. If `highPriorityKeywords` changes, the `remember` block will be executed and a new derived state object will be created and remembered in place of the old one. As the filtering to calculate `highPriorityTasks` can be expensive, it should only be executed when any of the lists change, not on every recomposition.

Furthermore, an update to the state produced by `derivedStateOf` doesn't cause the composable where it's declared to recompose, Compose only recomposes those composables where its returned state is read, inside `LazyColumn` in the example.

The code also assumes that `highPriorityKeywords` changes considerably less frequently than `todoTasks`. If that wasn't the case, the code could use `remember(todoTasks, highPriorityKeywords)` instead of `derivedStateOf`.

snapshotFlow: convert Compose's State into Flows

Use `snapshotFlow` (/reference/kotlin/androidx/compose/runtime/package-summary#snapshotFlow(kotlin.Function0)) to convert `State<T>` (/reference/kotlin/androidx/compose/runtime/State) objects into a cold Flow. `snapshotFlow` runs its block when collected and emits the result of the `State` objects read in it. When one of the `State` objects read inside the `snapshotFlow` block mutates, the Flow will emit the new value to its collector if the new value is not equal to (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-any/equals.html) the previous emitted value (this behavior is similar to that of `Flow.distinctUntilChanged` (https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/distinct-until-changed.html)
).

The following example shows a side effect that records when the user scrolls past the first item in a list to analytics:

```
val listState = rememberLazyListState()

LazyColumn(state = listState) {
    // ...
}

LaunchedEffect(listState) {
    snapshotFlow { listState.firstVisibleItemIndex }
        .map { index -> index > 0 }
        .distinctUntilChanged()
        .filter { it == true }
        .collect {
            MyAnalyticsService.sendScrolledPastFirstItemEvent()
        }
}
```

In the code above, `listState.firstVisibleItemIndex` is converted to a Flow that can benefit from the power of Flow's operators.

Restarting effects

Some effects in Compose, like `LaunchedEffect`, `produceState`, or `DisposableEffect`, take a variable number of arguments, keys, that are used to cancel the running effect and start a new one with the new keys.

The typical form for these APIs is:

```
EffectName(restartIfThisKeyChanges, orThisKey, orThisKey, ...) { block }
```

Due to the subtleties of this behavior, problems can occur if the parameters used to restart the effect are not the right ones:

- Restarting effects less than they should could cause bugs in your app.
- Restarting effects more than they should could be inefficient.

As a rule of thumb, mutable and immutable variables used in the effect block of code should be added as parameters to the effect composable. Apart from those, more parameters can be added to force restarting the effect. If the change of a variable shouldn't cause the effect to restart, the variable should be wrapped in `rememberUpdatedState`

(#rememberupdatedstate). If the variable never changes because it's wrapped in a `remember` with no keys, you don't need to pass the variable as a key to the effect.

Key Point: variables used in an effect should be added as a parameter of the effect composable, or use `rememberUpdatedState`.

In the `DisposableEffect` code shown above, the effect takes as a parameter the `lifecycleOwner` used in its block, because any change to them should cause the effect to restart.

```
@Composable
fun HomeScreen(
    lifecycleOwner: LifecycleOwner = LocalLifecycleOwner.current,
    onStart: () -> Unit, // Send the 'started' analytics event
    onStop: () -> Unit // Send the 'stopped' analytics event
) {
    // These values never change in Composition
    val currentOnStart by rememberUpdatedState(onStart)
    val currentOnStop by rememberUpdatedState(onStop)

    DisposableEffect(lifecycleOwner) {
        val observer = LifecycleEventObserver { _, event ->
            /* ... */
        }

        lifecycleOwner.lifecycle.addObserver(observer)
        onDispose {
            lifecycleOwner.lifecycle.removeObserver(observer)
        }
    }
}
```

`currentOnStart` and `currentOnStop` are not needed as `DisposableEffect` keys, because their value never change in Composition due to the usage of `rememberUpdatedState`. If you don't pass `lifecycleOwner` as a parameter and it changes, `HomeScreen` recomposes, but the `DisposableEffect` isn't disposed of and restarted. That causes problems because the wrong `lifecycleOwner` is used from that point onward.

Constants as keys

You can use a constant like `true` as an effect key to make it **follow the lifecycle of the call site**. There are valid use cases for it, like the `LaunchedEffect` example shown above. However, before doing that, think twice and make sure that's what you need.