State and Jetpack Compose

State in an app is any value that can change over time. This is a very broad definition and encompasses everything from a Room database to a variable on a class.

Jetpack Compose: State

All Android apps display state to the user. A few examples of state in Android apps:

- A Snackbar that shows when a network connection can't be established.
- A blog post and associated comments.
- Ripple animations on buttons that play when a user clicks them.
- Stickers that a user can draw on top of an image.

Jetpack Compose helps you be explicit about where and how you store and use state in an Android app. This guide focuses on the connection between state and composables, and on the APIs that Jetpack Compose offers to work with state more easily.

State and composition

Compose is declarative and as such the only way to update it is by calling the same composable with new arguments. These arguments are representations of the UI state. Any time a state is updated a *recomposition* takes place. As a result, things like `TextField` don't automatically update like they do in imperative XML based views. A composable has to explicitly be told the new state in order for it to update accordingly.

```
@Composable
private fun HelloContent() {
    Column(modifier = Modifier.padding(16.dp)) {
        Text(
            text = "Hello!",
            modifier = Modifier.padding(bottom = 8.dp),
            style = MaterialTheme.typography.bodyMedium
        )
        OutlinedTextField(
            value = "",
```

```
            onValueChange = { },
            label = { Text("Name") }
        )
    }
}
```

e/snippets/src/main/java/com/example/compose/snippets/state/StateOverviewSnippets.kt#L51-L65)

If you run this and try to enter text, you'll see that nothing happens. That's because the `TextField` doesn't update itself—it updates when its `value` parameter changes. This is due to how composition and recomposition work in Compose.

Key Term: Composition: a description of the UI built by Jetpack Compose when it executes composables.

Initial composition: creation of a Composition by running composables the first time.

Recomposition: re-running composables to update the Composition when data changes.

To learn more about initial composition and recomposition, see Thinking in Compose (/jetpack/compose/mental-model).

State in composables

Composable functions can use the `remember` (/reference/kotlin/androidx/compose/runtime/package-summary#remember(kotlin.Function0)) API to store an object in memory. A value computed by `remember` is stored in the Composition during initial composition, and the stored value is returned during recomposition. `remember` can be used to store both mutable and immutable objects.

Note: `remember` stores objects in the Composition, and forgets the object when the composable that called `remember` is removed from the Composition.

`mutableStateOf` (/reference/kotlin/androidx/compose/runtime/package-summary#mutableStateOf(kotlin.Any,androidx.compose.runtime.SnapshotMutationPolicy))

creates an observable `MutableState<T>` (/reference/kotlin/androidx/compose/runtime/MutableState), which is an observable type integrated with the compose runtime.

```
interface MutableState<T> : State<T> {
    override var value: T
}
```

Any changes to `value` schedules recomposition of any composable functions that read `value`. In the case of `ExpandingCard`, whenever `expanded` changes, it causes `ExpandingCard` to be recomposed.

There are three ways to declare a `MutableState` object in a composable:

- `val mutableState = remember { mutableStateOf(default) }`
- `var value by remember { mutableStateOf(default) }`
- `val (value, setValue) = remember { mutableStateOf(default) }`

These declarations are equivalent, and are provided as syntax sugar for different uses of state. You should pick the one that produces the easiest-to-read code in the composable you're writing.

The `by` delegate syntax requires the following imports:

```
import androidx.compose.runtime.getValue
import androidx.compose.runtime.setValue
```

You can use the remembered value as a parameter for other composables or even as logic in statements to change which composables are displayed. For example, if you don't want to display the greeting if the name is empty, use the state in an `if` statement:

```
@Composable
fun HelloContent() {
    Column(modifier = Modifier.padding(16.dp)) {
        var name by remember { mutableStateOf("") }
```

```
        if (name.isNotEmpty()) {
            Text(
                text = "Hello, $name!",
                modifier = Modifier.padding(bottom = 8.dp),
                style = MaterialTheme.typography.bodyMedium
            )
        }
        OutlinedTextField(
            value = name,
            onValueChange = { name = it },
            label = { Text("Name") }
        )
    }
}
```

e/snippets/src/main/java/com/example/compose/snippets/state/StateOverviewSnippets.kt#L70-L87)

While `remember` helps you retain state across recompositions, the state is not retained across configuration changes. For this, you must use `rememberSaveable`. `rememberSaveable` automatically saves any value that can be saved in a `Bundle`. For other values, you can pass in a custom saver object.

Caution: Using mutable objects such as `ArrayList<T>` or `mutableListOf()` as state in Compose causes your users to see incorrect or stale data in your app. Mutable objects that are not observable, such as ArrayList or a mutable data class, are not observable by Compose and don't trigger a recomposition when they change. Instead of using non-observable mutable objects, the recommendation is to use an observable data holder such as `State<List<T>>` and the immutable `listOf()`.

Other supported types of state

Compose doesn't require that you use `MutableState<T>` to hold state; it supports other observable types. Before reading another observable type in Compose, you must convert it to a `State<T>` so that composables can automatically recompose when the state changes.

Compose ships with functions to create `State<T>` (/reference/kotlin/androidx/compose/runtime/State) from common observable types used in Android apps. Before using these integrations, add the appropriate artifact(s) (/jetpack/androidx/releases/compose-runtime#declaring_dependencies) as outlined below:

- `Flow` (https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/-flow/index.html) : `collectAsStateWithLifecycle()` (/reference/kotlin/androidx/lifecycle/compose/package-summary#extension-functions)

  `collectAsStateWithLifecycle()` (/reference/kotlin/androidx/lifecycle/compose/package-summary#extension-functions) collects values from a `Flow` (https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/-flow/index.html) in a lifecycle-aware manner, allowing your app to save unneeded app resources. It represents the latest emitted value via Compose `State` (/reference/kotlin/androidx/compose/runtime/State). Use this API as the recommended way to collect flows on Android apps.

★ **Note:** To learn more about collecting flows safely in Android with `collectAsStateWithLifecycle()` API , you can read this blog post (https://medium.com/androiddevelopers/consuming-flows-safely-in-jetpack-compose-cde014d0d5a3) .

The following dependency (/jetpack/androidx/releases/lifecycle) is required in the `build.gradle` file (it should be 2.6.0-beta01 or newer):

Kotlin (#kotlin) Groovy (#groovy)

```
dependencies {
      ...
      implementation "androidx.lifecycle:lifecycle-runtime-compose:2.6.0
}
```

- `Flow` (https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/-flow/index.html)

: `collectAsState()` (/reference/kotlin/androidx/compose/runtime/package-summary#(kotlinx.coroutines.flow.StateFlow).collectAsState(kotlin.coroutines.CoroutineContext))

`collectAsState` is similar to `collectAsStateWithLifecycle`, because it also collects values from a `Flow` and transforms it into Compose `State` (/reference/kotlin/androidx/compose/runtime/State).

Use `collectAsState` for platform-agnostic code instead of `collectAsStateWithLifecycle`, which is Android-only.

Additional dependencies are not required for `collectAsState`, because it is available in `compose-runtime`.

- `LiveData` (/reference/kotlin/androidx/compose/runtime/livedata/package-summary): `observeAsState()` (/reference/kotlin/androidx/compose/runtime/livedata/package-summary#(androidx.lifecycle.LiveData).observeAsState(kotlin.Any))

  `observeAsState()` starts observing this `LiveData` (/reference/kotlin/androidx/lifecycle/LiveData) and represents its values via `State` (/reference/kotlin/androidx/compose/runtime/State).

  The following dependency (/jetpack/androidx/releases/compose-runtime) is required in the `build.gradle` file:

Kotlin (#kotlin) Groovy (#groovy)

```
dependencies {
      ...
      implementation "androidx.compose.runtime:runtime-livedata:1.4.2"
}
```

- `RxJava2` (/reference/kotlin/androidx/compose/runtime/rxjava2/package-summary): `subscribeAsState()` (/reference/kotlin/androidx/compose/runtime/rxjava2/package-summary#extension-functions)

  `subscribeAsState()` are extension functions that transform RxJava2's reactive streams (e.g. `Single` (http://reactivex.io/RxJava/2.x/javadoc/2.0.8/io/reactivex/Single.html), `Observable` (http://reactivex.io/RxJava/2.x/javadoc/2.0.8/io/reactivex/Observable.html), `Completable` (http://reactivex.io/RxJava/2.x/javadoc/2.0.8/io/reactivex/Completable.html)) into Compose `State` (/reference/kotlin/androidx/compose/runtime/State).

The following dependency (/jetpack/androidx/releases/compose-runtime) is required in the `build.gradle` file:

Kotlin (#kotlin) Groovy (#groovy)

```
dependencies {
      ...
      implementation "androidx.compose.runtime:runtime-rxjava2:1.4.2"
}
```

- `RxJava3` (/reference/kotlin/androidx/compose/runtime/rxjava3/package-summary): `subscribeAsState()` (/reference/kotlin/androidx/compose/runtime/rxjava3/package-summary#extension-functions)

  `subscribeAsState()` are extension functions that transform RxJava3's reactive streams (e.g. `Single` (http://reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Single.html), `Observable` (http://reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Observable.html), `Completable` (http://reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Completable.html)) into Compose `State` (/reference/kotlin/androidx/compose/runtime/State).

  The following dependency (/jetpack/androidx/releases/compose-runtime) is required in the `build.gradle` file:

Kotlin (#kotlin) Groovy (#groovy)

```
dependencies {
      ...
      implementation "androidx.compose.runtime:runtime-rxjava3:1.4.2"
}
```

Key Point: Compose automatically recomposes from reading **`State`** objects. If you use another observable type such as **`LiveData`** in Compose, you should convert it to **`State`** before reading it. Make sure that type conversion happens in a composable, using a composable extension function like **`LiveData<T>.observeAsState()`**.

Note: You are not limited to these integrations. You can build an extension function for Jetpack Compose that reads other observable types. If your app uses a custom observable class, convert it to produce `State<T>` using the **`produceState`** (/reference/kotlin/androidx/compose/runtime/package-summary#produceState(kotlin.Any,kotlin.coroutines.SuspendFunction1)) API.

See the implementation of the builtins for examples of how to do this: collectAsStateWithLifecycle (https://cs.android.com/androidx/platform/frameworks/support/+/androidx-main:lifecycle/lifecycle-runtime-compose/src/main/java/androidx/lifecycle/compose/FlowExt.kt;l=168?q=collectAsStateWithLifecycle). Any object that allows Jetpack Compose to subscribe to every change can be converted to `State<T>` and read in a Composable.

Stateful versus stateless

A composable that uses `remember` to store an object creates internal state, making the composable *stateful*. `HelloContent` is an example of a stateful composable because it holds and modifies its `name` state internally. This can be useful in situations where a caller doesn't need to control the state and can use it without having to manage the state themselves. However, composables with internal state tend to be less reusable and harder to test.

A *stateless* composable is a composable that doesn't hold any state. An easy way to achieve stateless is by using state hoisting (/jetpack/compose/state#state-hoisting).

As you develop reusable composables, you often want to expose both a stateful and a stateless version of the same composable. The stateful version is convenient for callers that don't care about the state, and the stateless version is necessary for callers that need to control or hoist the state.

State hoisting

State hoisting in Compose is a pattern of moving state to a composable's caller to make a composable stateless. The general pattern for state hoisting in Jetpack Compose is to replace the state variable with two parameters:

- **`value: T`**: the current value to display

- `onValueChange: (T) -> Unit`: an event that requests the value to change, where `T` is the proposed new value

However, you are not limited to `onValueChange`. If more specific events are appropriate for the composable you should define them using lambdas like `ExpandingCard` does with `onExpand` and `onCollapse`.

State that is hoisted this way has some important properties:

- **Single source of truth:** By moving state instead of duplicating it, we're ensuring there's only one source of truth. This helps avoid bugs.
- **Encapsulated:** Only stateful composables can modify their state. It's completely internal.
- **Shareable:** Hoisted state can be shared with multiple composables. If you wanted to read `name` in a different composable, hoisting would allow you to do that.
- **Interceptable:** callers to the stateless composables can decide to ignore or modify events before changing the state.
- **Decoupled:** the state for the stateless `ExpandingCard` may be stored anywhere. For example, it's now possible to move `name` into a `ViewModel`.

In the example case, you extract the `name` and the `onValueChange` out of `HelloContent` and move them up the tree to a `HelloScreen` composable that calls `HelloContent`.

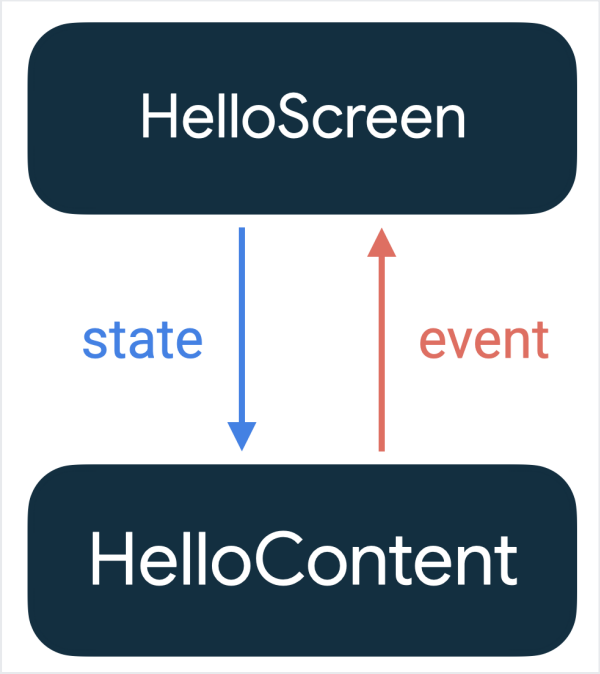
```
@Composable
fun HelloScreen() {
    var name by rememberSaveable { mutableStateOf("") }

    HelloContent(name = name, onNameChange = { name = it })
}

@Composable
fun HelloContent(name: String, onNameChange: (String) -> Unit) {
    Column(modifier = Modifier.padding(16.dp)) {
        Text(
            text = "Hello, $name",
            modifier = Modifier.padding(bottom = 8.dp),
            style = MaterialTheme.typography.bodyMedium
        )
        OutlinedTextField(value = name, onValueChange = onNameChange, label =
    }
}
```

/snippets/src/main/java/com/example/compose/snippets/state/StateOverviewSnippets.kt#L93-L110)

By hoisting the state out of `HelloContent`, it's easier to reason about the composable, reuse it in different situations, and test. `HelloContent` is decoupled from how its state is stored. Decoupling means that if you modify or replace `HelloScreen`, you don't have to change how `HelloContent` is implemented.



The pattern where the state goes down, and events go up is called a *unidirectional data flow*. In this case, the state goes down from `HelloScreen` to `HelloContent` and events go up from `HelloContent` to `HelloScreen`. By following unidirectional data flow, you can decouple composables that display state in the UI from the parts of your app that store and change state.

Key Point: When hoisting state, there are three rules to help you figure out where state should go:

1. State should be hoisted to at *least* the **lowest common parent** of all composables that use the state (read).
2. State should be hoisted to at *least* the **highest level it may be changed** (write).
3. If **two states change in response to the same events** they should be **hoisted together.**

You can hoist state higher than these rules require, but underhoisting state makes it difficult or impossible to follow unidirectional data flow.

See the Where to hoist state (/jetpack/compose/state-hoisting) page to learn more.

Restoring state in Compose

The `rememberSaveable` (/reference/kotlin/androidx/compose/runtime/saveable/package-summary#rememberSaveable(kotlin.Array,androidx.compose.runtime.saveable.Saver,kotlin.String,kotlin.Function0)) API behaves similarly to `remember` because it retains state across recompositions, and also across activity or process recreation using the saved instance state mechanism. For example, this happens, when the screen is rotated.

Ways to store state

All data types that are added to the `Bundle` are saved automatically. If you want to save something that cannot be added to the `Bundle`, there are several options.

Parcelize

The simplest solution is to add the `@Parcelize` (https://github.com/Kotlin/KEEP/blob/master/proposals/extensions/android-parcelable.md) annotation to the object. The object becomes parcelable, and can be bundled. For example, this code makes a parcelable `City` data type and saves it to the state.

```
@Parcelize
data class City(val name: String, val country: String) : Parcelable

@Composable
fun CityScreen() {
    var selectedCity = rememberSaveable {
        mutableStateOf(City("Madrid", "Spain"))
    }
}
```

snippets/src/main/java/com/example/compose/snippets/state/StateOverviewSnippets.kt#L116-L124)

MapSaver

If for some reason `@Parcelize` is not suitable, you can use `mapSaver` to define your own rule for converting an object into a set of values that the system can save to the `Bundle`.

```
data class City(val name: String, val country: String)

val CitySaver = run {
    val nameKey = "Name"
    val countryKey = "Country"
    mapSaver(
        save = { mapOf(nameKey to it.name, countryKey to it.country) },
        restore = { City(it[nameKey] as String, it[countryKey] as String) }
    )
}

@Composable
fun CityScreen() {
    var selectedCity = rememberSaveable(stateSaver = CitySaver) {
        mutableStateOf(City("Madrid", "Spain"))
    }
}
snippets/src/main/java/com/example/compose/snippets/state/StateOverviewSnippets.kt#L130-L146)
```

ListSaver

To avoid needing to define the keys for the map, you can also use `listSaver` and use its indices as keys:

```
data class City(val name: String, val country: String)

val CitySaver = listSaver<City, Any>(
    save = { listOf(it.name, it.country) },
    restore = { City(it[0] as String, it[1] as String) }
)

@Composable
fun CityScreen() {
    var selectedCity = rememberSaveable(stateSaver = CitySaver) {
        mutableStateOf(City("Madrid", "Spain"))
    }
}
snippets/src/main/java/com/example/compose/snippets/state/StateOverviewSnippets.kt#L153-L165)
```

State holders in Compose

Simple state hoisting can be managed in the composable functions itself. However, if the amount of state to keep track of increases, or the logic to perform in composable functions arises, it's a good practice to delegate the logic and state responsibilities to other classes: **state holders**.

Key Term: State holders manage logic and state of composables.

Note that in other materials, state holders are also called *hoisted state objects*.

See the state hoisting in Compose (/jetpack/compose/state-hoisting) documentation or, more generally, the State holders and UI State (/topic/architecture/ui-layer/stateholders) page in the architecture guide to learn more.

Retrigger remember calculations when keys change

The `remember` (/reference/kotlin/androidx/compose/runtime/package-summary#remember(kotlin.Any,kotlin.Any,kotlin.Any,kotlin.Function0)) API is frequently used together with `MutableState` (/reference/kotlin/androidx/compose/runtime/MutableState):

```
var name by remember { mutableStateOf("") }
```
snippets/src/main/java/com/example/compose/snippets/state/StateOverviewSnippets.kt#L172-L172)

Here, using the `remember` function makes the `MutableState` value survive recompositions.

In general, `remember` takes a `calculation` lambda parameter. When `remember` is first run, it invokes the `calculation` lambda and stores its result. During recomposition, `remember` returns the value that was last stored.

Apart from caching state, you can also use `remember` to store any object or result of an operation in the Composition that is expensive to initialize or calculate. You might not want to repeat this calculation in every recomposition. An example is creating this `ShaderBrush` (/reference/kotlin/androidx/compose/ui/graphics/ShaderBrush) object, which is an expensive operation:

```
val brush = remember {
    ShaderBrush(
        BitmapShader(
            ImageBitmap.imageResource(res, avatarRes).asAndroidBitmap(),
            Shader.TileMode.REPEAT,
            Shader.TileMode.REPEAT
        )
    )
}
```

snippets/src/main/java/com/example/compose/snippets/state/StateOverviewSnippets.kt#L181-L189)

`remember` stores the value until it leaves the Composition. However, there is a way to invalidate the cached value. The `remember` API also takes a `key` or `keys` parameter. *If any of these keys change, the next time the function recomposes*, `remember` *invalidates the cache and executes the calculation lambda block again*. This mechanism gives you control over the lifetime of an object in the Composition. The calculation remains valid until the inputs change, instead of until the remembered value leaves the Composition.

The following examples show how this mechanism works.

In this snippet, a `ShaderBrush` (/reference/kotlin/androidx/compose/ui/graphics/ShaderBrush) is created and used as the background paint of a `Box` composable. `remember` stores the `ShaderBrush` (/reference/kotlin/androidx/compose/ui/graphics/ShaderBrush) instance because it is expensive to recreate, as explained earlier. `remember` takes `avatarRes` as the `key1` parameter, which is the selected background image. If `avatarRes` changes, the brush recomposes with the new image, and reapplies to the `Box`. This can occur when the user selects another image to be the background from a picker.

```
@Composable
private fun BackgroundBanner(
    @DrawableRes avatarRes: Int,
    modifier: Modifier = Modifier,
    res: Resources = LocalContext.current.resources
```

```
) {
    val brush = remember(key1 = avatarRes) {
        ShaderBrush(
            BitmapShader(
                ImageBitmap.imageResource(res, avatarRes).asAndroidBitmap(),
                Shader.TileMode.REPEAT,
                Shader.TileMode.REPEAT
            )
        )
    }

    Box(
        modifier = modifier.background(brush)
    ) {
        /* ... */
    }
}
```

snippets/src/main/java/com/example/compose/snippets/state/StateOverviewSnippets.kt#L194-L215)

In the next snippet, state is hoisted to a plain state holder class (/topic/architecture/ui-layer/stateholders#choose_between_a_viewmodel_and_plain_class_for_a_state_holder) `MyAppState`. It exposes a `rememberMyAppState` function to initialize an instance of the class using `remember`. Exposing such functions to create an instance that survives recompositions is a common pattern in Compose. The `rememberMyAppState` function receives `windowSizeClass` (/reference/kotlin/androidx/compose/material3/windowsizeclass/WindowSizeClass), which serves as the `key` parameter for `remember`. If this parameter changes, the app needs to recreate the plain state holder class with the latest value. This may occur if, for example, the user rotates the device.

```
@Composable
private fun rememberMyAppState(
    windowSizeClass: WindowSizeClass
): MyAppState {
    return remember(windowSizeClass) {
        MyAppState(windowSizeClass)
    }
}

@Stable
class MyAppState(
```

```
    private val windowSizeClass: WindowSizeClass
) { /* ... */ }
```

snippets/src/main/java/com/example/compose/snippets/state/StateOverviewSnippets.kt#L219-L231)

Note: For more information about plain state holder classes, see the Plain state holder class as state owner (/jetpack/compose/state-hoisting#plain-state) documentation, or the State holders and UI State (/topic/architecture/ui-layer/stateholders) documentation in the Architecture guide.

Compose uses the class's equals (/reference/java/lang/Object#equals(java.lang.Object)) implementation to decide if a key has changed and invalidate the stored value.

Note: At first glance, using `remember` with keys might seem similar to using other Compose APIs, like `derivedStateOf` (/reference/kotlin/androidx/compose/runtime/package-summary#derivedStateOf(kotlin.Function0)). See the Jetpack Compose — When should I use derivedStateOf? (https://medium.com/androiddevelopers/jetpack-compose-when-should-i-use-derivedstateof-63ce7954c11b) blog post to learn about the difference.

Store state with keys beyond recomposition

The `rememberSaveable` (/reference/kotlin/androidx/compose/runtime/saveable/package-summary#rememberSaveable(kotlin.Array,androidx.compose.runtime.saveable.Saver,kotlin.String,kotlin.Function0)) API is a wrapper around `remember` that can store data in a `Bundle` (/reference/android/os/Bundle). This API allows state to survive not only recomposition, but also activity recreation and system-initiated process death. `rememberSaveable` receives `input` parameters for the same purpose that `remember` receives `keys`. *The cache is invalidated when any of the inputs change*. The next time the function recomposes, `rememberSaveable` re-executes the calculation lambda block.

Note: There is a difference in API naming that you should note. In the `remember` API, you use the parameter name `keys`, and in `rememberSaveable` you use `inputs` for the same purpose. If any of these parameters changes, the cached value is invalidated.

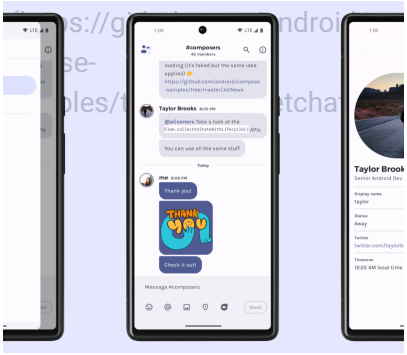
In the following example, `rememberSaveable` stores `userTypedQuery` until `typedQuery` changes:

```
var userTypedQuery by rememberSaveable(typedQuery, stateSaver = TextFieldValu
    mutableStateOf(
        TextFieldValue(text = typedQuery, selection = TextRange(typedQuery.le
    )
}
snippets/src/main/java/com/example/compose/snippets/state/StateOverviewSnippets.kt#L238-L242)
```

Learn more

To learn more about state and Jetpack Compose, consult the following additional resources.

Samples

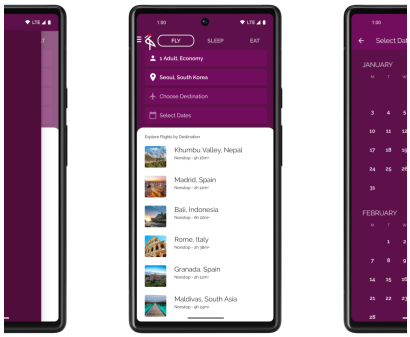


GITHUB

Jetchat sample

(https://github.com/android/compose-samples/tree/main/Jetchat)

Jetchat is a sample chat app built with Jetpack Compose. To try out this sample app, use the latest…

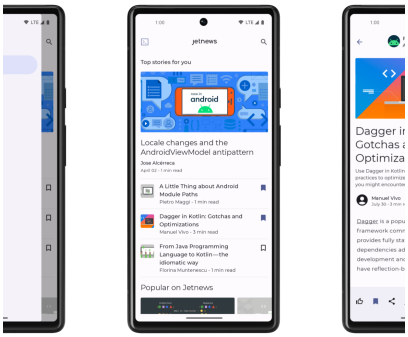


GITHUB

Crane sample

(https://github.com/android/compose-samples/tree/main/Crane)

Crane is a travel app part of the Material Studies built with Jetpack Compose. The goal of the sample is to…



GITHUB

Jetnews sample

(https://github.com/android/compose-samples/tree/main/JetNews)

Jetnews is a sample news reading app, built with Jetpack Compose. The goal of the sample is to…

More

Codelabs

- Using State in Jetpack Compose (https://codelabs.developers.google.com/codelabs/jetpack-compose-state/index.html?index=..%2F..index#0)

Videos

- A Compose state of mind (https://www.youtube.com/watch?v=rmv2ug-wW4U)

Blogs

- Effective State Management for `TextField` in Compose (https://medium.com/androiddevelopers/effective-state-management-for-textfield-in-compose-d6e5b070fbe5)

Last updated 2023-07-05 UTC.