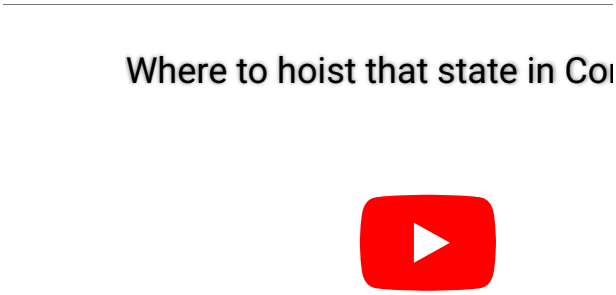
Where to hoist state

In a Compose application, where you hoist UI state (/topic/architecture/ui-layer#define-ui-state) depends on whether UI logic or business logic requires it. This document lays out these two main scenarios.

Best practice

You should hoist UI state to the **lowest common ancestor** between all the composables that read and write it. You should keep state closest to where it is consumed. From the state owner, expose to consumers immutable state and events to modify the state.



The lowest common ancestor can also be outside of the Composition. For example, when hoisting state in a `ViewModel` because business logic is involved.

This page explains this best practice in detail and a caveat to keep in mind.

Types of UI state and UI logic

Below there are definitions for types of UI state and logic that are used throughout this document.

UI state

UI state (/topic/architecture/ui-layer#define-ui-state) is the property that describes the UI. There are two types of UI state:

- **Screen UI state** is *what* you need to display on the screen. For example, a `NewsUiState` class can contain the news articles and other information needed to render the UI. This state is usually connected with other layers of the hierarchy because it contains app data.
- **UI element state** refers to properties intrinsic to UI elements that influence how they are rendered. A UI element may be shown or hidden and may have a certain font, font

size, or font color. In Android Views, the View manages this state itself as it is inherently stateful, exposing methods to modify or query its state. An example of this are the `get` (/reference/android/widget/TextView#getText()) and `set` (/reference/android/widget/TextView#setText(java.lang.CharSequence)) methods of the `TextView` (/reference/android/widget/TextView) class for its text. In Jetpack Compose, the state is external to the composable, and you can even hoist it out of the immediate vicinity of the composable into the calling composable function or a state holder. An example of this is `ScaffoldState` (/reference/kotlin/androidx/compose/material/ScaffoldState) for the `Scaffold` (/reference/kotlin/androidx/compose/material/package-summary#Scaffold(androidx.compose.ui.Modifier,androidx.compose.material.ScaffoldState,kotlin.Function0,kotlin.Function0,kotlin.Function1,kotlin.Function0,androidx.compose.material.FabPosition,kotlin.Boolean,kotlin.Function1,kotlin.Boolean,androidx.compose.ui.graphics.Shape,androidx.compose.ui.unit.Dp,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,kotlin.Function1)) composable.

Logic

Logic in an application can be either business logic or UI logic:

- **Business logic** is the implementation of product requirements for app data. For example, bookmarking an article in a news reader app when the user taps the button. This logic to save a bookmark to a file or database is usually placed in the domain or data layers. The state holder usually delegates this logic to those layers by calling the methods they expose.
- **UI logic** is related to *how* to display UI state on the screen. For example, obtaining the right search bar hint when the user has selected a category, scrolling to a particular item in a list, or the navigation logic to a particular screen when the user clicks a button.

UI logic

When UI logic (/topic/architecture/ui-layer#logic-types) needs to read or write state, you should scope the state to the UI, following its lifecycle. To achieve this, you should hoist the state at the correct level in a composable function. Alternatively, you can do so in a plain state holder class (/topic/architecture/ui-layer/stateholders#ui-logic), also scoped to the UI lifecycle.

Below is a description of both solutions and explanation of when to use which.

Composables as state owner

Having UI logic and UI element state in composables is a good approach if the state and logic is simple. You can leave your state internal to a composable or hoist as required.

No state hoisting needed

Hoisting state isn't always required. State can be kept internal in a composable when no other composable need to control it. In this snippet, there is a composable that expands and collapses on tap:

```
@Composable
fun ChatBubble(
    message: Message
) {
    var showDetails by rememberSaveable { mutableStateOf(false) } // Define t

    ClickableText(
        text = AnnotatedString(message.content),
        onClick = { showDetails = !showDetails } // Apply simple UI logic
    )

    if (showDetails) {
        Text(message.timestamp)
    }
}
se/snippets/src/main/java/com/example/compose/snippets/state/StateHoistingSnippets.kt#L57-L71)
```

The variable `showDetails` is the internal state for this UI element. It's only read and modified in this composable and the logic applied to it is very simple. Hoisting the state in this case therefore wouldn't bring much benefit, so you can leave it internal. Doing so makes this composable the owner and single source of truth of the expanded state.

Key Point: Keeping UI element state internal to composable functions is acceptable. This is a good solution if the state and logic you apply to it is simple and other parts of the UI hierarchy don't need the state. For example, this is usually the case of animation state.

Hoisting within composables

If you need to share your UI element state with other composables and apply UI logic to it in different places, you can hoist it higher in the UI hierarchy. This also makes your composables more reusable and easier to test.

The following example is a chat app that implements two pieces of functionality:

- The `JumpToBottom` button scrolls the messages list to the bottom. The button performs UI logic on the list state.
- The `MessagesList` list scrolls to the bottom after the user sends new messages. UserInput performs UI logic on the list state.

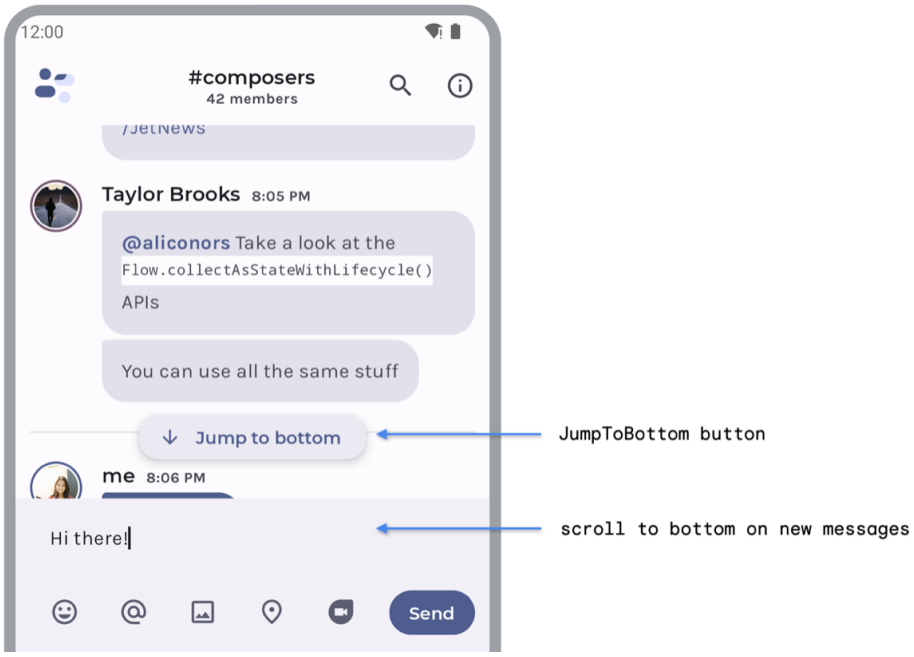


Figure 1. Chat app with a JumpToBottom button and scroll to bottom on new messages

The composable hierarchy is as follows:

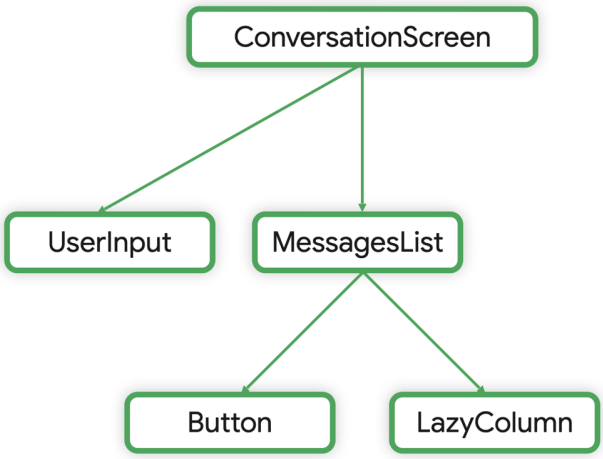


Figure 2. Chat composable tree

The `LazyColumn` (/reference/kotlin/androidx/compose/foundation/lazy/package-summary#LazyColumn(androidx.compose.ui.Modifier,androidx.compose.foundation.lazy.LazyListState,androidx.compose.foundation.layout.PaddingValues,kotlin.Boolean,androidx.compose.foundation.layout.Arrangement.Vertical,androidx.compose.ui.Alignment.Horizontal,androidx.compose.foundation.gestures.FlingBehavior,kotlin.Boolean,kotlin.Function1)) state is hoisted to the conversation screen so the app can perform UI logic and read the state from all composables that require it:

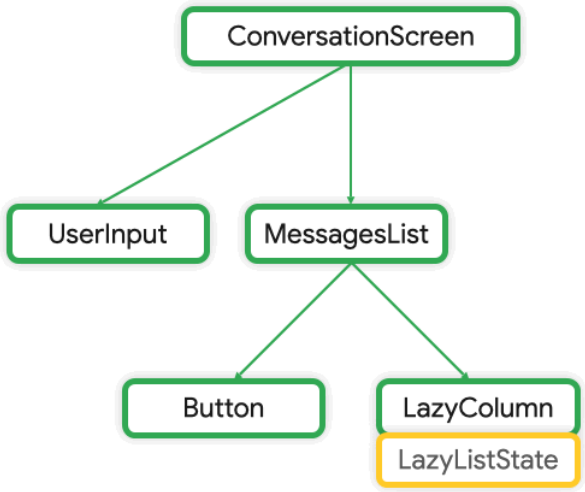


Figure 3. Hoisting LazyColumn state from the LazyColumn to the ConversationScreen

So finally the composables are:

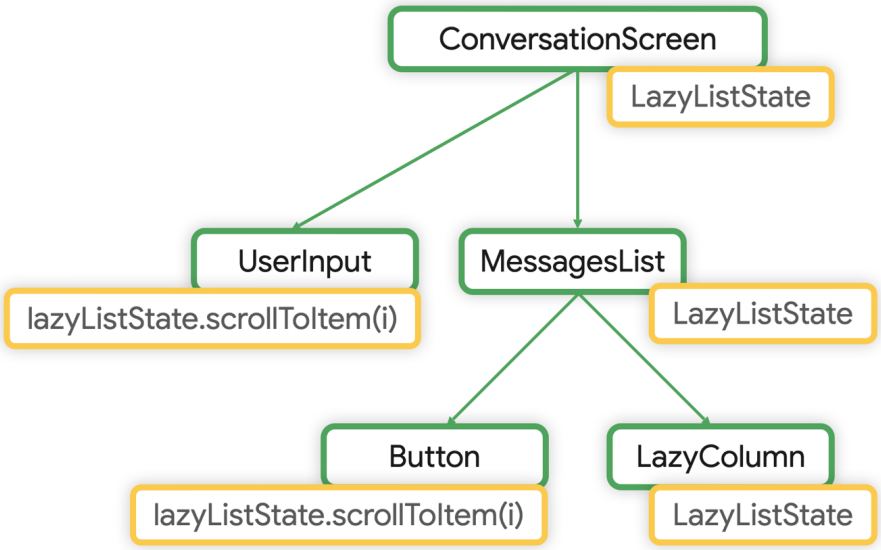


Figure 4. Chat composable tree with LazyListState hoisted to ConversationScreen

The code is as follows:

```
@Composable
private fun ConversationScreen(/*...*/) {
    val scope = rememberCoroutineScope()

    val lazyListState = rememberLazyListState() // State hoisted to the Conve

    MessagesList(messages, lazyListState) // Reuse same state in MessageList

    UserInput(
        onMessageSent = { // Apply UI logic to lazyListState
            scope.launch {
                lazyListState.scrollToItem(0)
            }
        },
    )
}

@Composable
private fun MessagesList(
    messages: List<Message>,
    lazyListState: LazyListState = rememberLazyListState() // LazyListState h
) {

    LazyColumn(
        state = lazyListState // Pass hoisted state to LazyColumn
    ) {
```

```
        items(messages, key = { message -> message.id }) { item ->
            Message(/*...*/)
        }
    }

    val scope = rememberCoroutineScope()

    JumpToBottom(onClicked = {
        scope.launch {
            lazyListState.scrollToItem(0) // UI logic being applied to lazyLi
        }
    })
}
```

e/snippets/src/main/java/com/example/compose/snippets/state/StateHoistingSnippets.kt#L85-L123)

`LazyListState` is hoisted as high as required for the UI logic that has to be applied. Since it is initialized in a composable function, it is stored in the Composition, following its lifecycle.

Note that `lazyListState` is defined in the `MessagesList` method, with the default value of `rememberLazyListState()`. This is a common pattern in Compose. It makes composables more reusable and flexible. You can then use the composable in different parts of the app which might not need to control the state. This is usually the case while testing or previewing a composable. This is exactly how LazyColumn defines its state.

Key Point: Hoist state to the lowest common ancestor and avoid passing it to composables that don't need it.

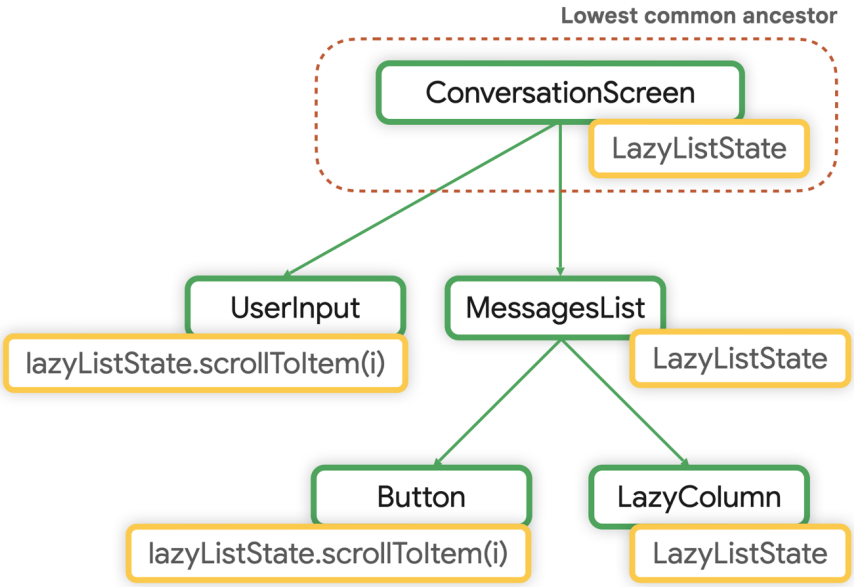


Figure 5. Lowest common ancestor for LazyListState is ConversationScreen

Plain state holder class as state owner

When a composable contains complex UI logic that involves one or multiple state fields of a UI element, it should delegate that responsibility to state holders (/topic/architecture/ui-layer/stateholders#ui-logic), like a plain state holder class. This makes the composable's logic more testable in isolation, and reduces its complexity. This approach favors the separation of concerns principle (https://en.wikipedia.org/wiki/Separation_of_concerns): **the composable is in charge of emitting UI elements, and the state holder contains the UI logic and UI element state**.

Plain state holder classes provide convenient functions to callers of your composable function, so they don't have to write this logic themselves.

These plain classes are created and remembered in the Composition. Because they follow the composable's lifecycle (/jetpack/compose/lifecycle), they can take types provided by the Compose library such as `rememberNavController()` (/reference/kotlin/androidx/navigation/compose/package-summary#rememberNavController(kotlin.Array)) or `rememberLazyListState()` (/reference/kotlin/androidx/compose/foundation/lazy/package-summary#rememberLazyListState(kotlin.Int,kotlin.Int)).

An example of this is the `LazyListState` (https://cs.android.com/androidx/platform/frameworks/support/+/androidx-main:compose/foundation/foundation/src/commonMain/kotlin/androidx/compose/foundation/lazy/LazyListState.kt;l=82?q=LazyListState.kt&ss=androidx%2Fplatform%2Fframeworks%2Fsupport) plain state holder class, implemented in Compose to control the UI complexity of `LazyColumn` (/reference/kotlin/androidx/compose/foundation/lazy/package-summary#LazyColumn(androidx.compose.ui.Modifier,androidx.compose.foundation.lazy.LazyListState,androidx.compose.foundation.layout.PaddingValues,kotlin.Boolean,androidx.compose.foundation.layout.Arrangement.Vertical,androidx.compose.ui.Alignment.Horizontal,androidx.compose.foundation.gestures.FlingBehavior,kotlin.Boolean,kotlin.Function1)) or `LazyRow` (/reference/kotlin/androidx/compose/foundation/lazy/package-summary#lazyrow).

```
// LazyListState.kt

@Stable
class LazyListState constructor(
    firstVisibleItemIndex: Int = 0,
    firstVisibleItemScrollOffset: Int = 0
) : ScrollableState {
    /**
     *   The holder class for the current scroll position.
     */
    private val scrollPosition = LazyListScrollPosition(
        firstVisibleItemIndex, firstVisibleItemScrollOffset
    )

    suspend fun scrollToItem(/*...*/) { /*...*/ }

    override suspend fun scroll() { /*...*/ }

    suspend fun animateScrollToItem() { /*...*/ }
}
```
snippets/src/main/java/com/example/compose/snippets/state/StateHoistingSnippets.kt#L129-L148)

`LazyListState` encapsulates the state of the `LazyColumn` (/reference/kotlin/androidx/compose/foundation/lazy/package-summary#LazyColumn(androidx.compose.ui.Modifier,androidx.compose.foundation.lazy.LazyListState,androidx.compose.foundation.layout.PaddingValues,kotlin.Boolean,androidx.compose.foundation.layout.Arrangement.Vertical,androidx.compose.ui.Alignment.Horizontal,androidx.compose.foundation.gestures.FlingBehavior,kotlin.Boolean,kotlin.Function1)) storing the `scrollPosition` for this UI element. It also exposes methods to modify the scroll position by for instance scrolling to a given item.

Note: This class is annotated as **`Stable`** (/reference/kotlin/androidx/compose/runtime/Stable). For more information on stability in Compose check this blog post (https://medium.com/androiddevelopers/jetpack-compose-stability-explained-79c10db270c8).

As you can see, **incrementing a composable's responsibilities increases the need for a state holder**. The responsibilities could be in UI logic, or just in the amount of state to keep track of.

Note: If plain state holder classes contain state you want to preserve after an Activity or process is recreated, use **`rememberSaveable`** and create a custom **`Saver`** for it.

Another common pattern is using a plain state holder class to handle the complexity of root composable functions in the app. You can use such a class to encapsulate app-level state like navigation state and screen sizing. A complete description of this can be found in the UI logic and its state holder page (/topic/architecture/ui-layer/stateholders#ui-logic).

Business logic

If composables and plain state holders classes are in charge of the UI logic and UI element state, a screen level state holder is in charge of the following tasks:

- Providing access to the business logic (/topic/architecture/ui-layer#logic-types) of the application that is usually placed in other layers of the hierarchy such as the business and data layers.
- Preparing the application data for presentation in a particular screen, which becomes the screen UI state.

ViewModels as state owner

The benefits (/topic/libraries/architecture/viewmodel#best-practices) of AAC ViewModels in Android development make them suitable for providing access to the business logic and preparing the application data for presentation on the screen.

Key Point: A `ViewModel` is just an implementation detail of a state holder with certain responsibilities. If you want to keep your project's module away from Android dependencies, you can rely on interfaces to make the implementation swappable in different contexts. For instance, you can use `ViewModel` in your Android-specific module, and simpler platform-agnostic implementations in other modules, like a plain state holder class.

When you hoist UI state in the `ViewModel`, you move it outside of the Composition.

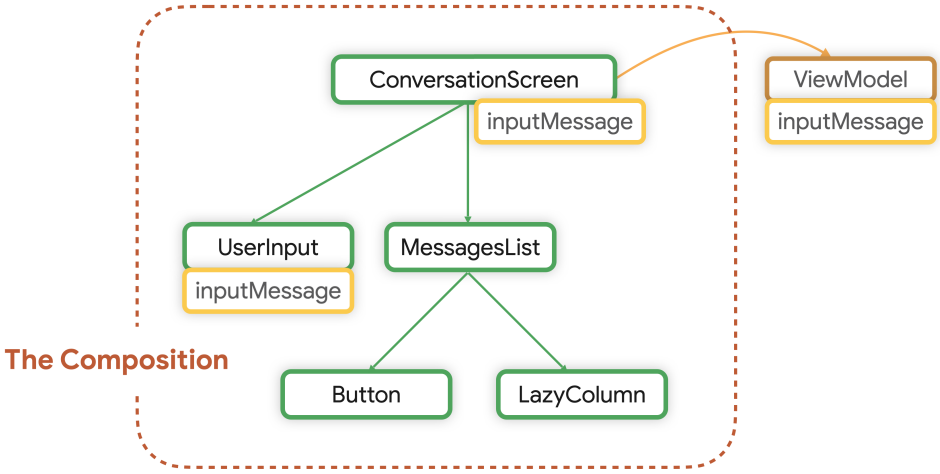


Figure 6. State hoisted to the ViewModel is stored outside of the Composition.

ViewModels aren't stored as part of the Composition. They're provided by the framework and they're scoped to a `ViewModelStoreOwner` (/reference/androidx/lifecycle/ViewModelStoreOwner) which can be an Activity, Fragment, navigation graph, or destination of a navigation graph. For more information on `ViewModel` scopes (/topic/libraries/architecture/viewmodel/viewmodel-apis) you can review the documentation.

Then, the `ViewModel` is the source of truth and **lowest common ancestor** for UI state.

Screen UI state

As per the definitions above, screen UI state is produced by applying business rules. Given that the screen level state holder is responsible for it, this means the screen UI state (/topic/architecture/ui-layer/stateholders#ui-state) is typically hoisted in the screen level state holder, in this case a `ViewModel`.

Consider the `ConversationViewModel` of a chat app and how it exposes the screen UI state and events to modify it:

```
class ConversationViewModel(
    channelId: String,
    messagesRepository: MessagesRepository
) : ViewModel() {

    val messages = messagesRepository
        .getLatestMessages(channelId)
        .stateIn(
            scope = viewModelScope,
            started = SharingStarted.WhileSubscribed(5_000),
            initialValue = emptyList()
        )

    // Business logic
    fun sendMessage(message: Message) { /* ... */ }
}
```

'snippets/src/main/java/com/example/compose/snippets/state/StateHoistingSnippets.kt#L158-L173)

Composables consume the screen UI state hoisted in the `ViewModel`. You should inject the `ViewModel` instance in your screen-level composables to provide access to business logic.

Note: You shouldn't pass `ViewModel` instances down to other composables. For more information, see the Architecture state holders documentation (/topic/architecture/ui-layer/stateholders#business-logic).

The following is an example of a `ViewModel` used in a screen-level composable. Here, the composable `ConversationScreen()` consumes the screen UI state hoisted in the `ViewModel`:

```
@Composable
private fun ConversationScreen(
    conversationViewModel: ConversationViewModel = viewModel()
) {

    val messages by conversationViewModel.messages.collectAsStateWithLifecycl

    ConversationScreen(
        messages = messages,
        onSendMessage = { message: Message -> conversationViewModel.sendMessa
    )
```

```
}

@Composable
private fun ConversationScreen(
    messages: List<Message>,
    onSendMessage: (Message) -> Unit
) {

    MessagesList(messages, onSendMessage)
    /* ... */
}
```

'snippets/src/main/java/com/example/compose/snippets/state/StateHoistingSnippets.kt#L177-L198)

Note: To use the `viewModel()` function, add the **`androidx.lifecycle:lifecycle-viewmodel-compose`** (/jetpack/androidx/releases/lifecycle) dependency to your `build.gradle` file. Learn more about this function in our documentation for working with other libraries (/jetpack/compose/libraries#viewmodel) in Compose.

Note: If ViewModels contain state that you want to preserve after system-initiated process recreation, use **`SavedStateHandle`** (/reference/androidx/lifecycle/SavedStateHandle) to persist it. For more information, see the Saving UI states page (/topic/libraries/architecture/saving-states).

Property drilling

"Property drilling" refers to passing data through several nested children components to the location where they're read.

A typical example of where property drilling can appear in Compose is when you inject the screen level state holder at the top level and pass down state and events to children composables. This might additionally generate an overload of composable functions signatures.

Even though exposing events as individual lambda parameters could overload the function signature, it maximizes the visibility of what the composable function responsibilities are. You can see what it does at a glance.

Property drilling is preferable over creating wrapper classes to encapsulate state and events in one place because this reduces the visibility of the composable responsibilities.

By not having wrapper classes you're also more likely to pass composables only the parameters they need, which is a best practice (/jetpack/compose/architecture#composable-parameters).

The same best practice applies if these events are navigation events, you can learn more about that in the navigation docs (/jetpack/compose/navigation#nav-calls-best-practices).

If you have identified a performance issue, you may also choose to defer reading of state. You can check the performance docs (/jetpack/compose/performance/bestpractices#defer-reads) to learn more.

UI element state

You can hoist UI element state to the screen level state holder if there is business logic that needs to read or write it.

Continuing the example of a chat app, the app displays user suggestions in a group chat when the user types `@` and a hint. Those suggestions come from the data layer and the logic to calculate a list of user suggestions is considered business logic. The feature looks like this:

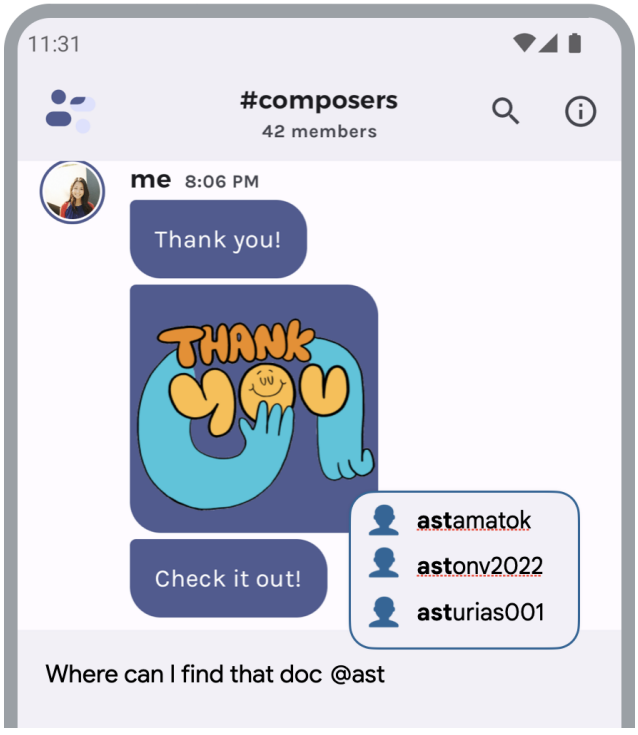


Figure 7. Feature that displays user suggestions in a group chat when the user types `@` and a hint

The `ViewModel` implementing this feature would look as follows:

```
class ConversationViewModel(/*...*/) : ViewModel() {

    // Hoisted state
    var inputMessage by mutableStateOf("")
        private set

    val suggestions: StateFlow<List<Suggestion>> =
        snapshotFlow { inputMessage }
            .filter { hasSocialHandleHint(it) }
            .mapLatest { getHandle(it) }
            .mapLatest { repository.getSuggestions(it) }
            .stateIn(
                scope = viewModelScope,
                started = SharingStarted.WhileSubscribed(5_000),
                initialValue = emptyList()
            )

    fun updateInput(newInput: String) {
        inputMessage = newInput
    }
}
```

'snippets/src/main/java/com/example/compose/snippets/state/StateHoistingSnippets.kt#L213-L233)

`inputMessage` is a variable storing the `TextField` (/reference/kotlin/androidx/compose/material/package-summary#TextField(kotlin.String,kotlin.Function1,androidx.compose.ui.Modifier,kotlin.Boolean,kotlin.Boolean,androidx.compose.ui.text.TextStyle,kotlin.Function0,kotlin.Function0,kotlin.Function0,kotlin.Function0,kotlin.Boolean,androidx.compose.ui.text.input.VisualTransformation,androidx.compose.foundation.text.KeyboardOptions,androidx.compose.foundation.text.KeyboardActions,kotlin.Boolean,kotlin.Int,androidx.compose.foundation.interaction.MutableInteractionSource,androidx.compose.ui.graphics.Shape,androidx.compose.material.TextFieldColors)) state. Every time the user types in new input, the app calls business logic to produce `suggestions`.

Note: if this variable wasn't needed for business logic as it is needed now to produce user suggestions, it shouldn't be hoisted to the screen level state holder. It should be defined and stored in the UI, closer to the composable function that needs it.

`suggestions` is screen UI state and is consumed from Compose UI by collecting from the `StateFlow`

(https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/-state-flow/)

.

Note: it's possible for a screen-level composable to have both a `ViewModel` that provides access to business logic AND a plain state holder class that manages its UI logic and UI elements' state.

Caveat

For some Compose UI element state, hoisting to the `ViewModel` might require special considerations. For example, some state holders of Compose UI elements expose methods to modify the state. Some of them might be suspend functions that trigger animations. These suspend functions can throw exceptions if you call them from a `CoroutineScope` (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-scope/) that is not scoped to the Composition.

Let's say the app drawer's content is dynamic and you need to fetch and refresh it from the data layer after it's closed. You should hoist the drawer state to the `ViewModel` so you can call both the UI and business logic on this element from the state owner.

However, calling `DrawerState` (/reference/kotlin/androidx/compose/material/DrawerState)'s `close()` (/reference/kotlin/androidx/compose/material/DrawerState#close()) method using the `viewModelScope` (/topic/libraries/architecture/coroutines#viewmodelscope) from Compose UI causes a runtime exception of type `IllegalStateException` (https://docs.oracle.com/javase/7/docs/api/java/lang/IllegalStateException.html) with a message reading "a `MonotonicFrameClock` (/reference/kotlin/androidx/compose/runtime/MonotonicFrameClock) is not available in this `CoroutineContext`" (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.coroutines/-coroutine-context/).

To fix this, use a `CoroutineScope` scoped to the Composition. It provides a `MonotonicFrameClock` in the `CoroutineContext` that is necessary for the suspend functions to work.

Warning: Calling some suspend functions exposed from Compose UI element state that trigger animations throw exceptions if called from a `CoroutineScope` that's not scoped to the Composition. For example, `LazyListState.animateScrollTo()`

(/reference/kotlin/androidx/compose/foundation/lazy/LazyListState#animateScrollToItem(kotlin.Int,kotlin.Int))
and **`DrawerState.close()`** (/reference/kotlin/androidx/compose/material/DrawerState#close()).

To fix this crash, switch the `CoroutineContext` of the coroutine in the `ViewModel` to one that is scoped to the Composition. It could look like this:

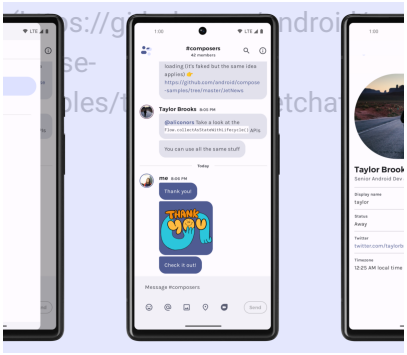
```
class ConversationViewModel(/*...*/) : ViewModel() {

    val drawerState = DrawerState(initialValue = DrawerValue.Closed)

    private val _drawerContent = MutableStateFlow(DrawerContent.Empty)
    val drawerContent: StateFlow<DrawerContent> = _drawerContent.asStateFlow(

    fun closeDrawer(uiScope: CoroutineScope) {
        viewModelScope.launch {
            withContext(uiScope.coroutineContext) { // Use instead of the def
                drawerState.close()
            }
            // Fetch drawer content and update state
            _drawerContent.update { content }
        }
    }
}

// in Compose
@Composable
private fun ConversationScreen(
    conversationViewModel: ConversationViewModel = viewModel()
) {
    val scope = rememberCoroutineScope()

    ConversationScreen(onCloseDrawer = { conversationViewModel.closeDrawer(ui
}
```

snippets/src/main/java/com/example/compose/snippets/state/StateHoistingSnippets.kt#L246-L272)

Learn more

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

Samples

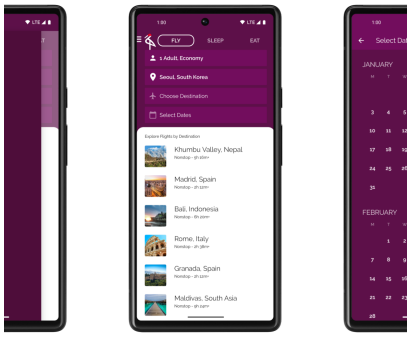


GITHUB

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…

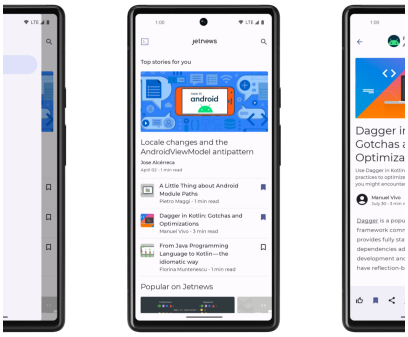


GITHUB

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

- State holders and state production in the UI Layer (https://youtu.be/pCX9wvu-Bq0)

Last updated 2023-06-20 UTC.