State Management in Jetpack Compose

Kaan Enes KAPICI
Senior Android Engineer
@TurkTelekom/INNOVA

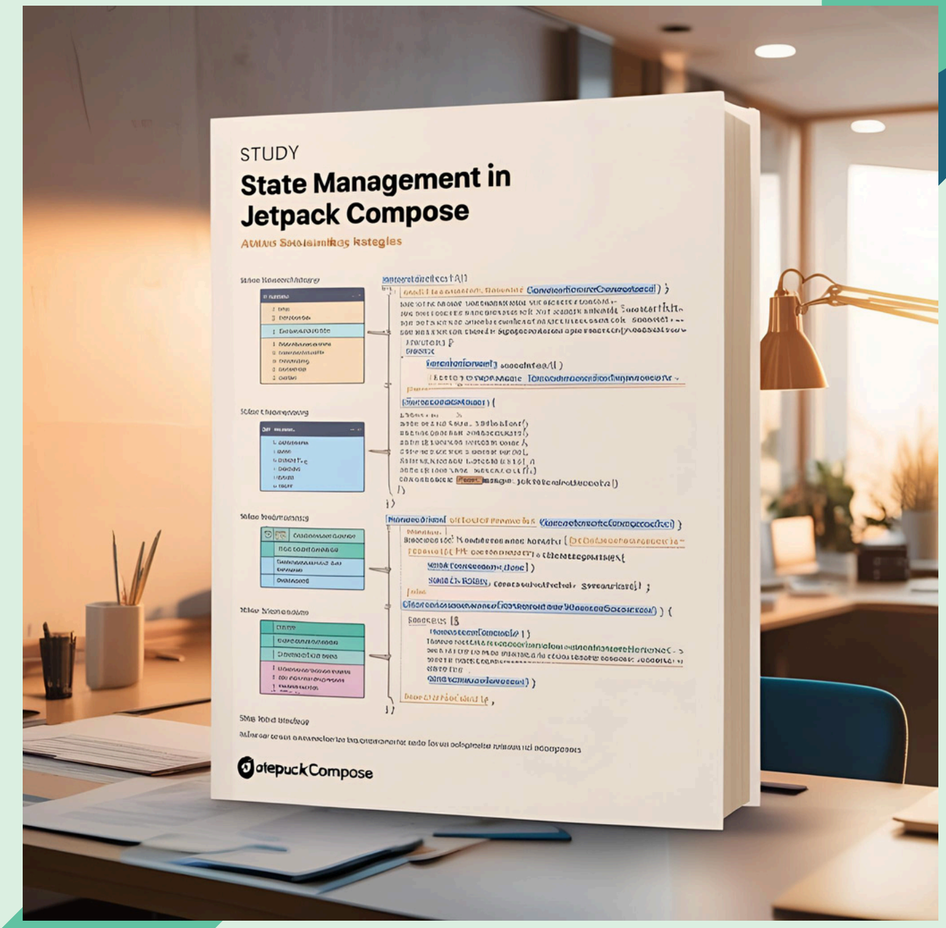
STATE BASICS

WHAT IS STATE?

State refers to any value that can change over time within an app.

WHAT IS EVENT?

User interactions/System changes that trigger state updates



DECLARATIVE UI AND RECOMPOSITION

Jetpack Compose adopts a declarative UI paradigm, meaning the UI is described in terms of its current state

Recomposition occurs when state changes, prompting Compose to re-execute composable functions with the new state, thereby updating the UI 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") }
        )
    }
}
```

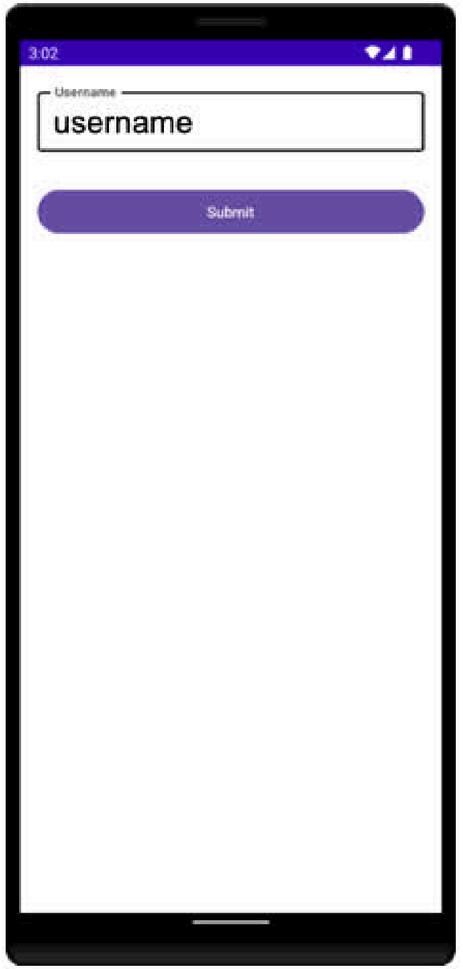
DECLARATIVE UI AND RECOMPOSITION

```
@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") }
        )
    }
}
```

- remember: Stores a single object in memory during recomposition.
- mutableStateOf: Creates an observable state holder that triggers recomposition when its value changes.
- derivedStateOf: Derives a new state based on other states, optimizing recompositions by recalculating only when necessary.

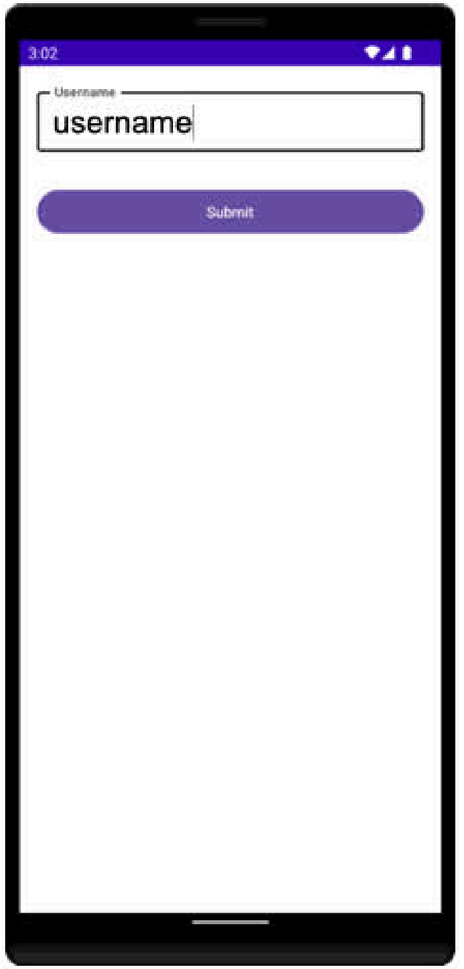
```
var username by
    remember { mutableStateOf("") }
val submitEnabled
    = isUsernameValid(username)
```

username	submitEnabled
	true
use	
us	true
u	true
	false



```
var username by
    remember { mutableStateOf("") }
val submitEnabled = remember {
    derivedStateOf { isUsernameValid(username) }
}
```

username	submitEnabled
userna	
usern	
user	
use	
us	
u	true
	false



REMEMBER(KEY1, KEY2 ...) { ... }

```
@Composable
fun Demo() {
    var activeKey by remember { mutableStateOf("A") }

    Column {
        Row {
            Button(onClick = { activeKey = "A" }) { Text("A") }
            Button(onClick = { activeKey = "B" }) { Text("B") }
        }
        Spacer(Modifier.height(12.dp))
        KeyedCounter(key = activeKey)
    }
}

@Composable
fun KeyedCounter(key: String) {

    var count by remember(key) { mutableStateOf(0) }

    Button(onClick = { count++ }) {
        Text("'$key' tık sayısı: $count")
    }
}
```

The key parameters of remember allow you to re-run the calculation within the block only when the keys change.

State	remember(key)
If the object parameter radically changes the UI behavior (url, id, page count)	✅ **KEY**
If the same reference does not change, only its subfields are updated (StateFlow value)	✅ **KEY**
Resource must be rebuilt (Animatable, CoroutineScope, Controller)	❌ **MOSTLY NO NEED**
Only "heavy" account caching (sort, filter)	✅ **KEY**

REMEMBERSAVEABLE

```
LazyColumn {
    items(
        books,
        key = { it.id }
    ) { book ->
        var checked by rememberSaveable(book.id) { mutableStateOf(false) }

        Row(
            verticalAlignment = Alignment.CenterVertically,
            modifier = Modifier
                .fillMaxWidth()
                .toggleable(
                    value = checked,
                    onValueChange = { checked = it }
                )
        ) {
            Checkbox(checked, null)
            Text(book.title)
        }
    }
}
```

rememberSaveable stores the state in a Bundle as long as it is called with the same key/input sequence:

Redrawings (recomposition)

Screen rotation / multi-window

Kill and restart in the background (process death)

restores data in all such cases. When the key (keys / inputs) changes, the cache is canceled and the block re-runs

OTHER SUPPORTED TYPES OF STATE

```
class ExampleViewModel : ViewModel() {

    private val _counter = MutableStateFlow(0)
    val counter: StateFlow<Int> = _counter.asStateFlow()

    fun incrementCounter() {
        _counter.value += 1
    }
}

@Composable
fun CounterScreen(viewModel: ExampleViewModel = viewModel()) {
    val counter by viewModel.counter.collectAsStateWithLifecycle()

    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        Text(
            text = "Counter: $counter",
            style = MaterialTheme.typography.headlineMedium
        )

        Spacer(modifier = Modifier.height(16.dp))

        Button(onClick = { viewModel.incrementCounter() }) {
            Text(text = "Increment")
        }
    }
}
```

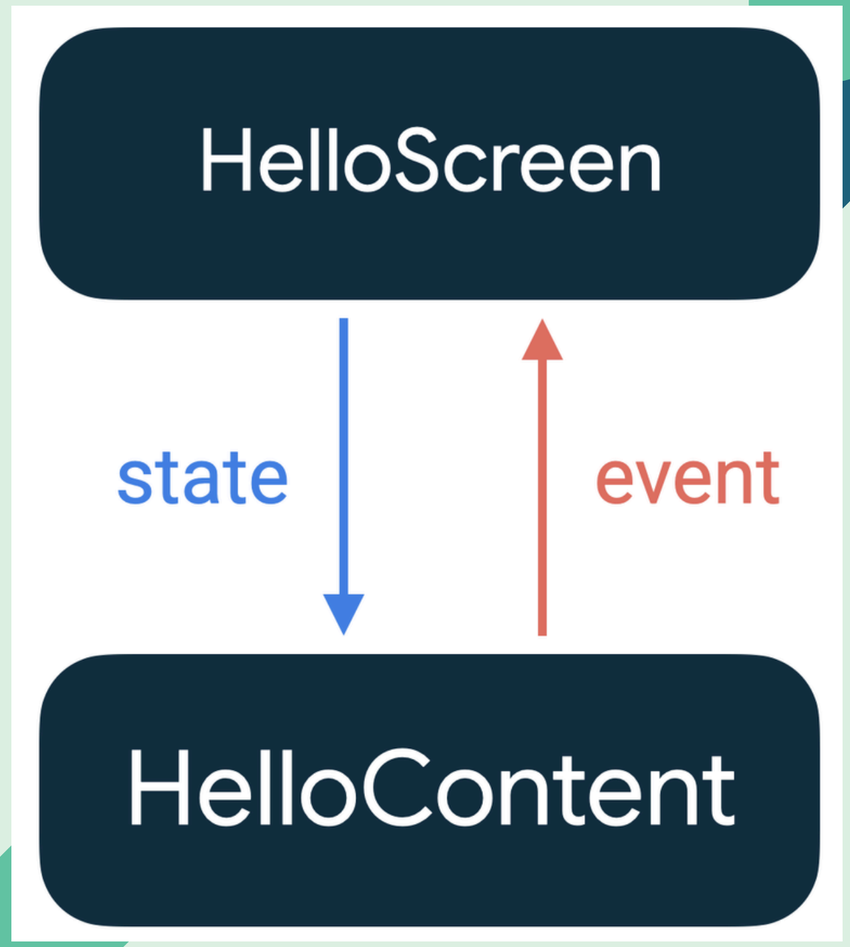
Flow: collectAsStateWithLifecycle() → Android Lifecycle Aware

Flow: collectAsState() → platform-agnostic code

STATE HOISTING

State hoisting involves moving state up to a common ancestor composable, enhancing reusability and testability.

This pattern separates stateless UI components from the stateful logic that manages their data.



STATE HOISTING

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 composables may be stored anywhere. For example, it's now possible to move name into a ViewModel.

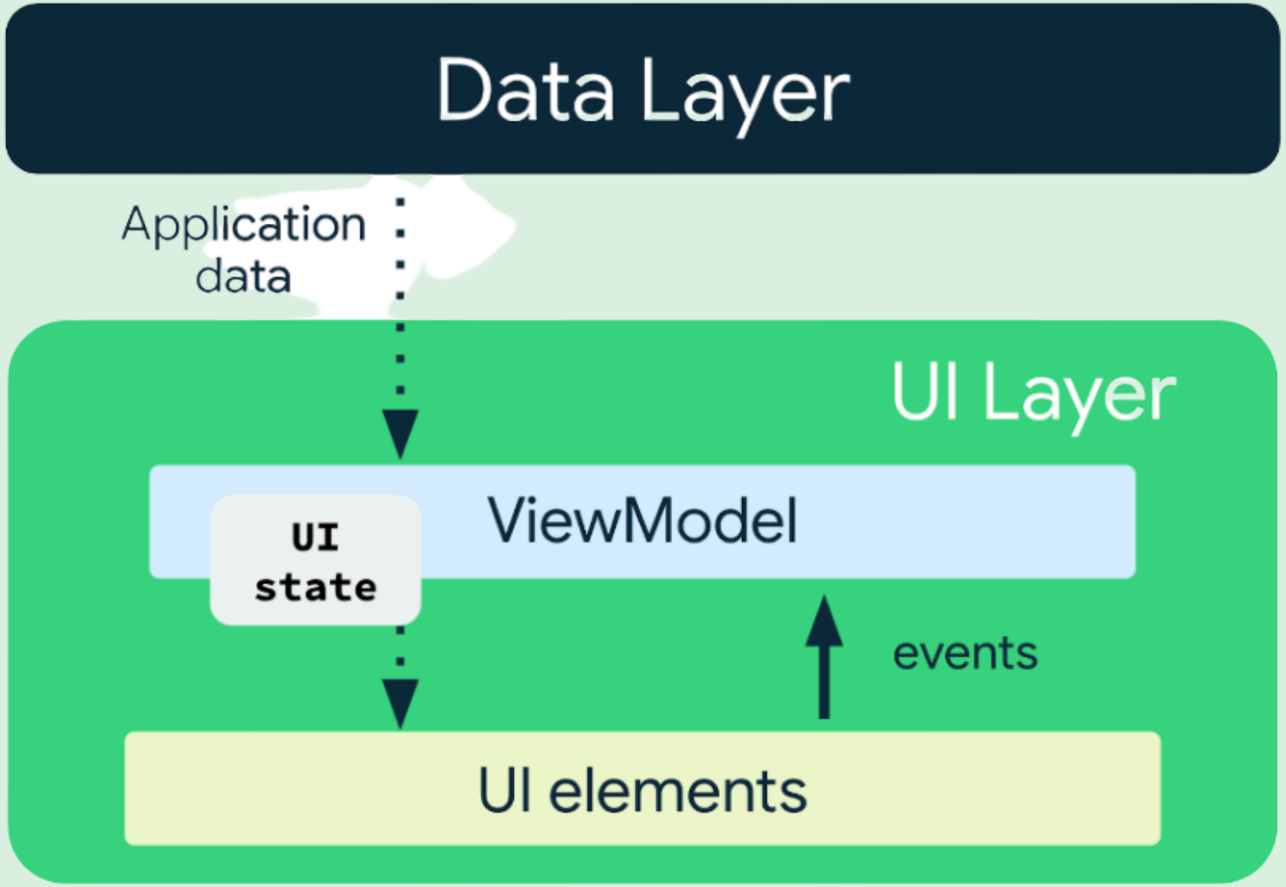
STATE HOISTING

```
@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 = { Text("Name") })
    }
}
```

STATE HOLDERS

State holder is a flat Kotlin class that contains the state shown in the UI and the business logic that updates that state.

To leave the UI layer "bare, just showing" and move it to pure classes that are testable, reusable and not lifecycle dependent.



```
@Composable
fun Counter(
    state: CounterStateHolder,
    modifier: Modifier = Modifier
) {
    Row(modifier, verticalAlignment = Alignment.CenterVertically) {
        IconButton(onClick = state::decrement) { Icon(Icons.Default.Remove, null) }
        Text(text = state.count.toString(), style = MaterialTheme.typography.headlineMedium)
        IconButton(onClick = state::increment) { Icon(Icons.Default.Add, null) }
    }
}
```

```
class CounterStateHolder {
    var count by mutableStateOf(0)
        private set

    fun increment() { count++ }
    fun decrement() { if (count > 0) count-- }
}
```

```
@Composable
fun CounterScreen() {
    val counterState = remember { CounterStateHolder() }
    Counter(state = counterState, modifier = Modifier.padding(16.dp))
}
```

RESTORING STATE IN COMPOSE

The rememberSaveable 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

Parcelize

MapSaver

ListSaver

PARCELIZE

The simplest solution is to add the @Parcelize 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"))
    }
}
```

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"))
    }
}
```

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"))
    }
}
```

SIDE EFFECTS

In Declarative UI you describe “how the screen will look”, but some things go outside the UI tree

Such as

writing to a database

making API calls

reading/writing files

displaying a Snackbar message

logging

```
1  LaunchedEffect(event) {
2          when (event) {
3              is UiEvent.ShowSnackbar -> {
4                  snackbarHostState.showSnackbar((event as UiEvent.ShowSnackbar).message)
5              }
6              else -> {}
7          }
8      }
```

SIDE EFFECTS

API	When
LaunchedEffect	• used for asynchronous operations that need to
SideEffect	• used to send data or updates to objects not managed by Compose
RememberCoroutineScope	• to launch coroutines in the background, triggered by user interaction, whose scope is connected to the UI.
DisposableEffect	• some side effects need to be cleaned up when they are no longer needed.(Observer, Broadcast Receiver, Socket Connection)
RememberUpdateState	• where we don't want the coroutine to restart even if some values used inside the effect change

REMEMBER UPDATE STATE

```kotlin
@Composable
fun CountdownTimer(
    totalTime: Int,
    onCountdownFinished: () -> Unit
) {
    // Her zaman güncel onCountdownFinished referansını tutar.
    val currentOnCountdownFinished by rememberUpdatedState(onCountdownFinished)

    LaunchedEffect(Unit) {
        var remainingTime = totalTime
        while (remainingTime > 0) {
            delay(1000)
            remainingTime--
        }
        // Süre bittiğinde en son callback çağrılır
        currentOnCountdownFinished()
    }

    Text("Kalan zaman: $totalTime saniye")
}
```

DISPOSABLE EFFECT

```
@Composable
fun BroadcastReceiverExample(context: Context) {
    DisposableEffect(Unit) {
        val intentFilter = IntentFilter(Intent.ACTION_BATTERY_CHANGED)
        val receiver = object : BroadcastReceiver() {
            override fun onReceive(ctx: Context?, intent: Intent?) {
                Log.d("Battery", "Battery changed!")
            }
        }
        context.registerReceiver(receiver, intentFilter)

        onDispose {
            context.unregisterReceiver(receiver) // Temizlik yapılıyor.
        }
    }
}
```

REMEMBER COROUTINE SCOPE

```
1  @Composable
2  fun CoroutineScopeExample() {
3      val scope = rememberCoroutineScope()
4      val snackbarHostState = remember { SnackbarHostState() }
5
6      Column {
7          Button(onClick = {
8              scope.launch {
9                  snackbarHostState.showSnackbar("Button clicked!")
10             }
11         }) {
12             Text("Click me")
13         }
14         SnackbarHost(snackbarHostState)
15     }
16 }
```

LAUNCHED EFFECT

```
@Composable
fun LaunchedEffectExample(userId: String, viewModel: UserViewModel) {
    LaunchedEffect(userId) {
        viewModel.loadUser(userId)
    }

    val user by viewModel.user.collectAsState()

    Text("User: ${user.name}")
}
```

SIDE EFFECT

```
1 @Composable
2 fun SideEffectExample(userName: String) {
3     SideEffect {
4         Analytics.trackScreenViewed("Profile: $userName")
5     }
6
7     Text("Welcome $userName")
8 }
```

Thanks for listening. Is there anything you'd like to ask?

If you want to reach me, you can scan the QR code.