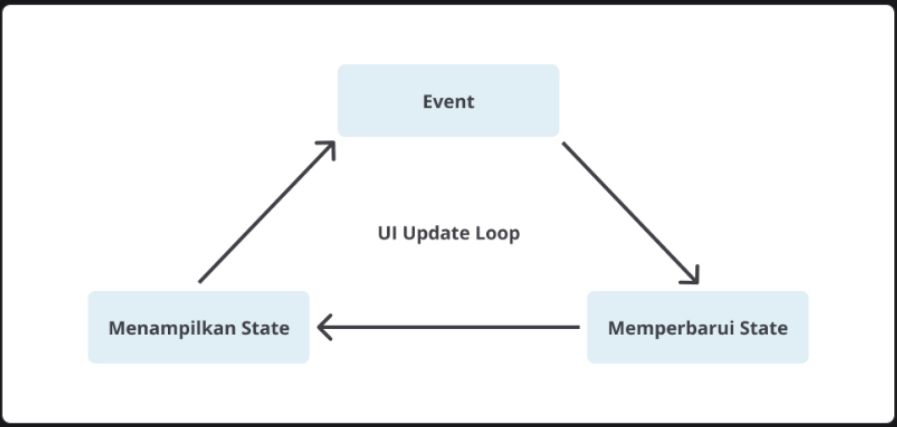
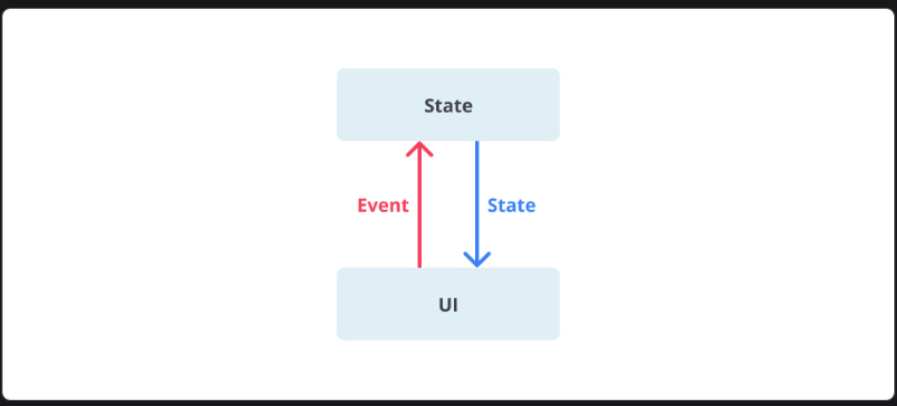
Rangkuman State pada Compose

- State adalah data yang ditampung pada suatu komponen. Data ini dapat berubah-ubah sepanjang waktu dan membuat UI juga berubah.
- Ada banyak contoh **penerapan State**, seperti
    - mengubah button menjadi aktif ketika input yang diberikan valid;
    - menentukan radio button yang dipilih;
    - menampilkan dropdown ketika action menu dipilih;
    - menampilkan loading ketika memuat data;
    - menentukan posisi scroll pada list; dan
    - menentukan halaman yang tampil ketika menu dipilih.
- Berikut gambaran umum **urutan untuk memperbarui UI**.

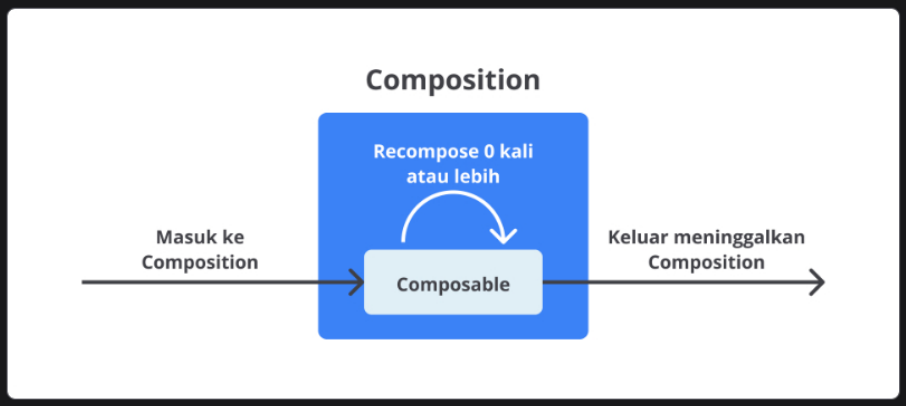


- **Event** adalah segala sesuatu yang dapat menyebabkan State berubah. Perubahan bisa dari dalam aplikasi seperti penekanan tombol maupun dari luar aplikasi seperti hasil sensor dan Web API.
- Inti dari mengatur state di Compose adalah tentang memahami State dan Event saling berinteraksi. Jetpack Compose menggunakan Unidirectional Data Flow untuk mengaturnya. **Unidirectional Data Flow (UDF)** adalah pattern yang mengalirkan Events ke atas dan State ke bawah.



- Dengan mengikuti pattern UDF tersebut, Anda akan mendapatkan beberapa manfaat berikut.
    - **Testability**: memisahkan state dengan UI membuat kedua komponen mudah untuk dites.
    - **State encapsulation**: proses pengubahan state hanya bisa dilakukan di satu tempat sehingga mengurangi potensi munculnya bug.
    - **UI consistency**: state yang dibuat langsung direfleksikan dalam bentuk UI sehingga membuat UI selalu konsisten dengan state yang didefinisikan.

- **mutableStateOf** berfungsi untuk melacak state. Ia akan mengubah nilai di dalamnya menjadi MutableState yang merupakan tipe observable di Compose. Compose akan selalu membaca nilai di dalamnya sehingga ketika ada perubahan data, UI juga akan dibuat ulang.
- **remember** berfungsi untuk menyimpan nilai tersebut ke dalam memori sehingga datanya tidak hilang ketika terjadi recomposition.
- **by remember** merupakan property delegation yang berfungsi untuk menangani proses getValue dan setValue di dalam remember.
- **rememberSaveable** berfungsi untuk menyimpan state dalam bentuk Bundle ketika terjadi configuration change.
- **Stateful** adalah jenis Composable Function yang menyimpan state di dalamnya. Hal ini bisa ditandai dengan adanya fungsi remember di dalamnya. Jenis ini kurang reusable dan lebih susah dites karena logika yang ada di dalamnya tidak dapat kita atur.
- **Stateless** adalah jenis Composable Function yang tidak menyimpan state di dalamnya. Jenis ini lebih fleksibel karena Anda dapat mengatur logika dari luar dan mudah dites.
- **State hoisting** merupakan *pattern* untuk memindahkan state ke parent atasnya (Composable pemanggilnya) supaya sebuah Composable menjadi stateless.
- Berikut beberapa manfaat State Hoisting.
    - **Single Source of Truth**: hanya ada sumber untuk mengubah data, yakni dari pemanggilnya.
    - **Shareable**: komponen bisa digunakan lebih dari satu Composable.
    - **Interceptable**: pemanggil bisa memilih untuk membiarkan atau menggunakan Event.
    - **Decoupled**: state bisa diletakkan terpisah dari Composable, misalnya di ViewModel.
- Untuk mengimplementasikan State Hoisting, langkah yang perlu Anda lakukan adalah mengubah variabel state menjadi dua parameter berikut.
    - **value: T**, nilai state yang ditampilkan.
    - **onEvent: (T) -> Unit**, lambda yang menunjukkan event yang terjadi.
- Berikut beberapa aturan yang bisa dijadikan acuan untuk menentukan lokasi State.
    - State setidaknya diletakkan di **level terendah** dari semua Composable yang **membaca state**.
    - State setidaknya diletakkan di **level tertinggi** dari Composable yang **mengubah state**.
    - Jika **dua state berubah ketika ada event yang sama**, mereka harus diletakkan di **level yang sama**.
- **Lifecycle Composable** sangatlah simpel. Pertama, kita masuk ke Composition, kemudian Recomposition 0 kali atau lebih, dan terakhir keluar dari Composition.



- **Side Effect** adalah segala sesuatu yang mengubah state dari luar scope Composable Function.
- **Side Effect API** berfungsi supaya Anda dapat mengontrol Side Effect supaya tetap aman dan memastikan beberapa hal berikut.
    - Memastikan Effect berjalan pada lifecycle yang tepat.

    - Memastikan Effect dibersihkan ketika keluar dari Composition.
    - Memastikan Suspend Function dibatalkan ketika keluar dari Composition.
    - Effect yang bergantung pada input yang bervariasi secara otomatis dibersihkan dan dijalankan lagi ketika nilainya berubah.

- Berikut beberapa Side Effect API yang bisa dimanfaatkan.

    - **launchedEffect**: digunakan ketika akan ingin melakukan aksi tertentu yang hanya dipanggil sekali ketika initial composition atau ketika parameter Key berubah. Selain itu, ia juga merupakan Coroutine Scope, sehingga Anda dapat memanggil suspend function di dalamnya.
    - **rememberUpdatedState**: digunakan untuk menandai suatu nilai supaya tidak ter-restart walaupun key berubah.
    - **rememberCoroutineScope**: membuat coroutine scope untuk menjalankan suspend function di dalam Composable Function yang aware dengan lifecycle compose.
    - **SideEffect**: digunakan untuk membuat state dari kode non Compose dan akan dipanggil setiap kali recomposition berhasil.
    - **DisposableEffect**: biasanya digunakan untuk membersihkan sesuatu ketika meninggalkan composition menggunakan **onDispose**.
    - **produceState**: Berfungsi untuk membuat non Compose State menjadi Compose State baru.
    - **derivedStateOf**: mengubah satu atau lebih State menjadi sebuah State baru.
    - **snapshotFlow**: mengonversi State pada Compose menjadi Flow.

- Macam-macam tipe State.

    - **UI element state**: hoisted state untuk elemen UI, seperti ScaffoldState.
    - **Screen/UI state**: state yang dibuat untuk menentukan komponen yang tampil di halaman, seperti loading, error, atau berhasil. State ini biasanya dibuat sendiri dan berhubungan dengan data yang akan ditampilkan.

- Macam-macam tipe logic.

    - **UI behavior logic atau UI logic**: logika yang berhubungan dengan tampilan UI, misalnya membuat tombol aktif atau tidak aktif dan menampilkan Snackbar atau Toast. Logika ini seharusnya tetap berada di Composable.
    - **Business Logic**: logika yang berhubungan dengan fungsi utama aplikasi, misalnya menyimpan keranjang, melakukan checkout, dan memproses pembayaran. Logika ini biasanya ada di layer data (repository), bukan di layer UI.

- Kemungkinan lokasi sebagai Single Source of Truth (SSoT) untuk memanajemen State.

    - **Composable**: untuk manajemen state elemen UI yang simpel.
    - **State Holder**: untuk manajemen state elemen UI yang kompleks. Berisi state elemen UI dan *UI logic*.
    - **Architecture Component ViewModel**: untuk dapat mengakses *business logic* sekaligus state UI atau Screen untuk menyiapkan data yang ditampilkan.

- **State Holder** merupakan class biasa berisi UI element State dan UI logic yang saling berkaitan.
- Berikut beberapa manfaat menerapkan State Holder.

    - Memusatkan semua perubahan State dalam satu tempat saja.
    - Kemungkinan untuk terjadinya out of sync menjadi sedikit.
    - Bisa dipakai lagi untuk komponen yang sama di tempat berbeda.
    - Mengurangi kompleksitas dari Composable.
    - Memudahkan untuk State Hoisting.

- **mutableStateOf** diperlukan untuk membaca data ketika terjadi perubahan. Karena terletak di luar Composable, ia tidak perlu dibarengi dengan remember.
- Apabila data berupa stream, gunakanlah beberapa extension berikut untuk mengubahnya menjadi State.
    - **Flow.collectAsState()**: mengubah Flow menjadi State. Tidak diperlukan dependency tambahan.
    - **LiveData.observeAsState()**: mengubah LiveData menjadi State. Membutuhkan dependency androidx.compose.runtime:runtime-livedata.
    - **Observable.subscribeAsState()**: mengubah Observable object dari RxJava2 atau RxJava3 menjadi State.
- Alasan pemilihan ViewModel daripada State Holder adalah karena ia memiliki beberapa kelebihan. Jika Anda tidak mendapatkan manfaat di bawah ini, gunakan saja State Holder.
    - Mempertahankan data ketika configuration change (seperti ketika device rotasi).
    - Terintegrasi dengan library Jetpack lainnya, seperti Hilt dan Navigation Component.
    - Disimpan di cache ketika di Navigation backstack dan dibersihkan ketika keluar.
- Berikut beberapa hal yang perlu diperhatikan ketika menggunakan ViewModel.
    - ViewModel memiliki *lifetime* yang lebih panjang daripada Composable. Untuk itu, jangan letakkan state yang menahan Composition (seperti ScaffoldState) pada ViewModel karena bisa menyebabkan *memory leak*.
    - Jangan kirimkan object ViewModel ke composable lainnya. Cukup kirimkan argument yang dibutuhkan saja. Hal ini penting supaya Recomposition efektif.
    - Pisahkan Stateless dan Stateful ViewModel. Hal ini akan memudahkan Anda untuk melakukan test dan preview.