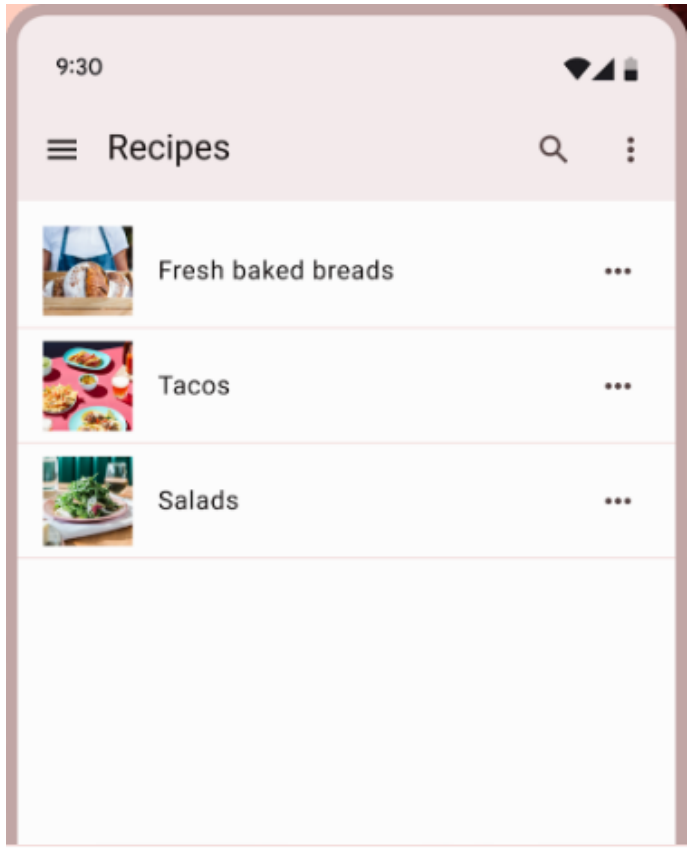
SCROLLABLE LIST

Tim Dosen Mobpro 1

D3 Rekayasa Perangkat Lunak Aplikasi
Fakultas Ilmu Terapan

DISPLAYING LIST



- Many apps need to display collections of items.. Use lists to help users find an item and act on it.
- Lists are optimized for reading comprehension. So, keep items short and easy to scan by placing elements in consistent locations across list items.
- Always order list items in logical ways (like alphabetical or numerical).

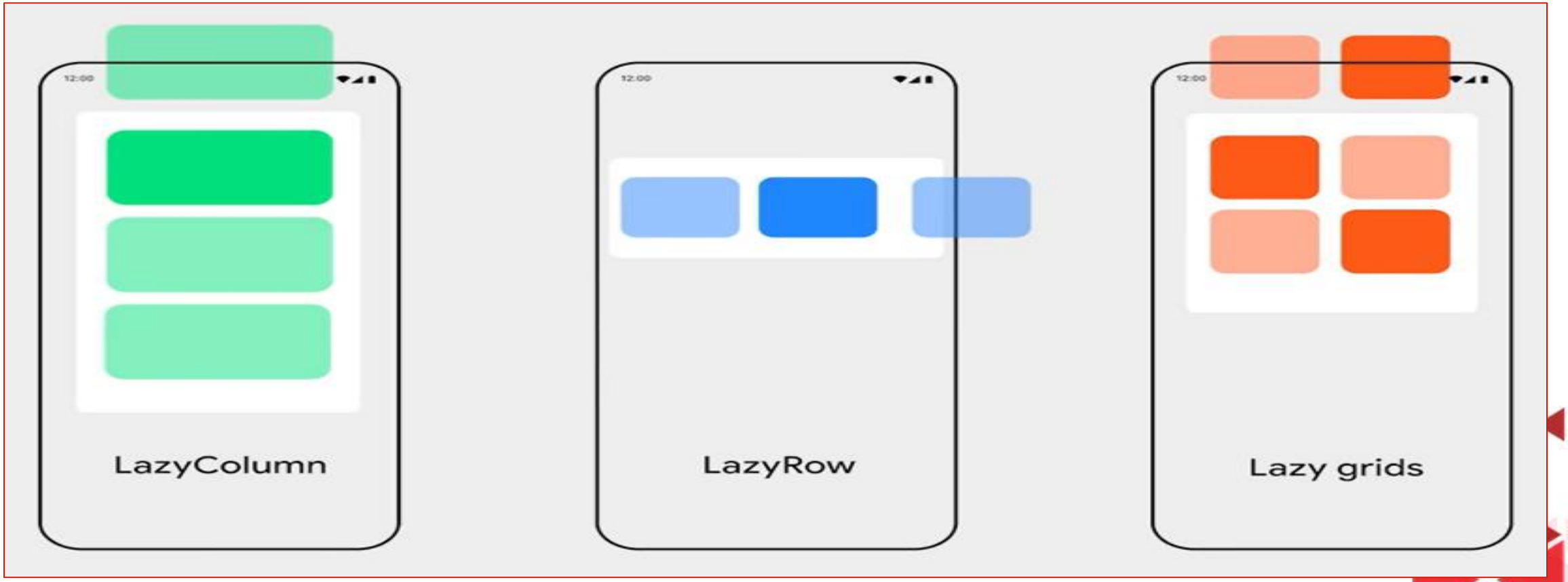
DISPLAYING LIST

- If you need to display a large number of items, using a layout such as Column can cause **performance issues**, since all the items will be composed and laid out whether or not they are visible.

- Compose provides a set of components which **only** compose and lay out items which are **visible** in the component's viewport. These components include LazyColumn and LazyRow.

- If you've used the RecyclerView widget in the Views UI framework (XML), these components follow the same set of principles, but with **less code**.

LISTS AND GRIDS



LAZY GRIDS

The lazy grid composables provide support for displaying items in a grid.

LazyVerticalGrid	LazyHorizontalGrid
LazyVerticalStaggeredGrid	LazyHorizontalStaggeredGrid

A Lazy vertical grid will display its items in a vertically scrollable container, spanned across **multiple columns**, while the Lazy horizontal grids will have the same behaviour on the horizontal axis.

The difference between a lazy staggered grid and a lazy grid is that the former can display items of **different size** widths or heights.

LAZY COLUMN – STEP 1

```
@Composable
fun ListItem(catatan: Catatan, onClick: () -> Unit) {
    Column(
        modifier = Modifier.clickable { onClick() }
        verticalArrangement = Arrangement.spacedBy(8.dp)
    ) {
        Text(text = catatan.judul)
        Text(text = catatan.catatan)
        Text(text = catatan.tanggal)
    }
}
```

LAZY COLUMN – STEP 2

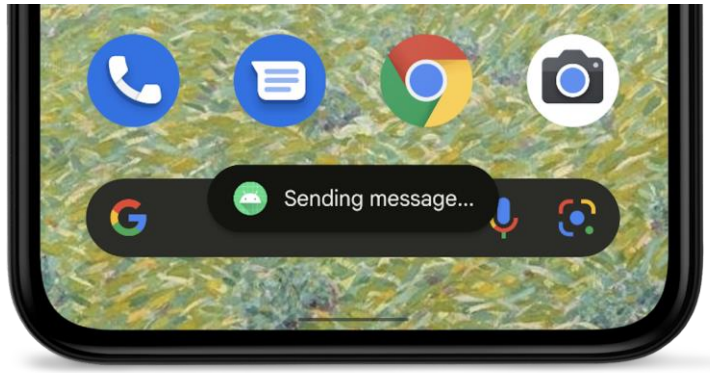
```
LazyColumn(
    modifier = modifier.fillMaxSize(),
    contentPadding = PaddingValues(bottom = 84.dp)
) {
    items(data) {
        ListItem(catatan = it) {
            val pesan = context.getString(R.string.x_diklik, it.judul)
            Toast.makeText(context, pesan, Toast.LENGTH_SHORT).show()
        }
        Divider()
    }
}
```

Add padding bottom if you are using FAB.

TOAST

- A toast provides **simple feedback** about an operation in a small popup. Toasts automatically disappear after a timeout.

- A toast only fills the amount of space required for the message and the current activity remains visible and interactive.

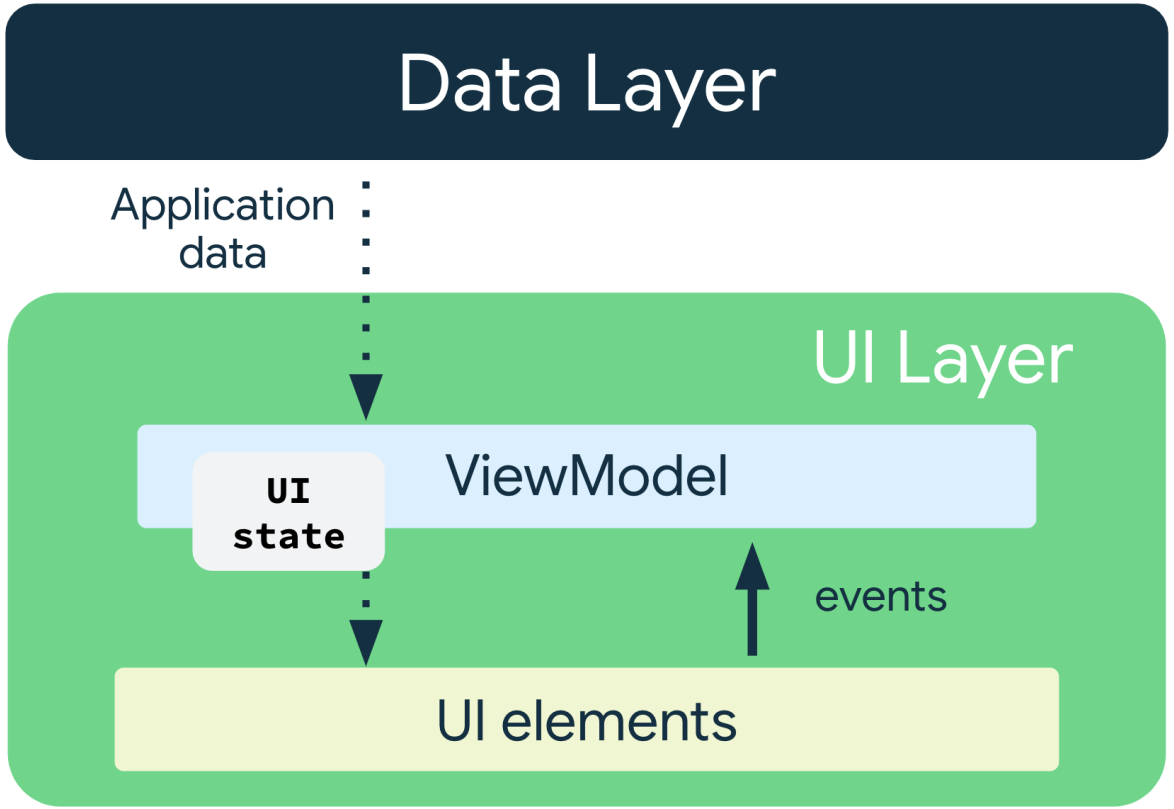


- If your app targets Android 12 (API level 31) or higher, its toast is limited to two lines of text and shows the application icon next to the text.

TOAST ALTERNATIVES

Component	Priority	User action
Snackbar	Low priority	Optional: Snackbars disappear automatically
Banner	Prominent, medium priority	Optional: Banners remain until dismissed by the user, or if the state that caused the banner is resolved
Dialog	Highest priority	Required: Dialogs block app usage until the user takes a dialog action or exits the dialog (if available)

APP ARCHITECTURE



The most important principle to follow is **separation of concerns**.

Each application should have at least two layers:

- The UI layer that displays application data on the screen.
- The data layer that contains the business logic of your app and exposes application data.

VIEW MODEL

- The ViewModel class is a **business logic** or screen level **state holder**. It exposes state to the UI and encapsulates related business logic.

- It caches state and persists it through configuration changes. So, your UI doesn't have to fetch data again when navigating between activities, or following configuration changes, such as when rotating the screen.

- The lifecycle of a ViewModel is tied directly to its scope. A ViewModel **remains in memory** until the ViewModelStoreOwner to which it is scoped disappears. In the case of a Navigation entry, this may occur when it's removed from the back stack.

HANDLE EMPTY STATES

Empty states occur when an item's content can't be shown. For example, a list without list items, or a search that returns no results. Although these states aren't typical, they should be designed to **prevent confusion**.

To handle empty states, you can:

- Displaying an empty state that consists of an image and a text tagline.
- Populating with starter content to allows users to begin using an app.
- Showing educational content to helps users understand your app.
- Displaying content that contains the best match to the user's query.

REFERENCES

- Android Basics with Compose
  https://developer.android.com/courses/android-basics-compose/course

- Lists and grids
  https://developer.android.com/jetpack/compose/lists

- Guide to app architecture
  https://developer.android.com/topic/architecture

- Toasts overview
  https://developer.android.com/guide/topics/ui/notifiers/toasts