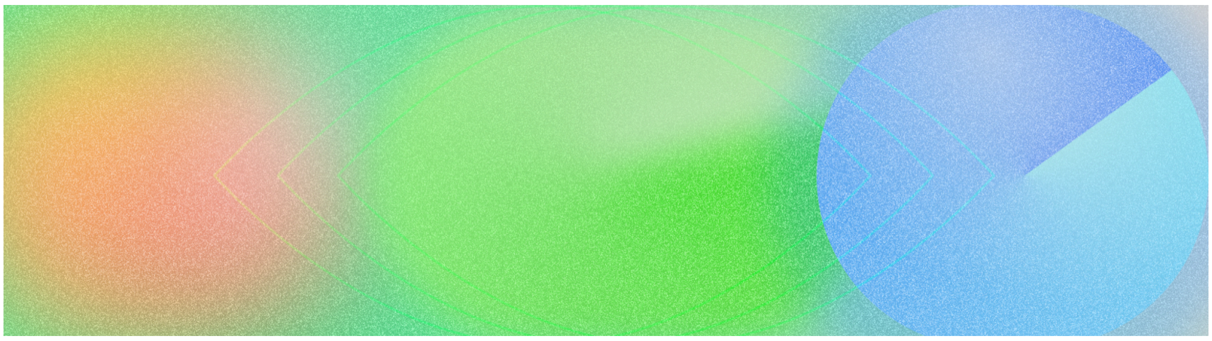
Accessibility

bookmark_border



According to a 2011 report by the World Health Organization (WHO) and the World Bank, approximately 15% of the global population–that is, about one in six people–experience a significant or temporary disability in their lifetime. Accessibility in design, then, is *fundamental* to creating an inclusive, usable, and high-quality app–it leads to the best results for users and can prevent costly rework. Android ships with a variety of features to help you build your app to support accessibility options by default.

Design for vision

Ensure your app's content is as legible as possible by checking color contrast and text sizing, and that components are visually comprehensible and easy to discern from each other.

Follow these guidelines to design for vision accessibility.

- To allow users to adjust the font size, specify font size in scalable pixels (sp)
- Don't make the body size any smaller than 12 sp. This guideline aligns with the Material typescale as a default.
- Ensure the contrast between the background and text is at least 4.5:1. Learn how to check color contrast.
- Use a 3:1 ratio between surfaces and non-text elements. For example, the ratio of a background to an icon would be 3:1.
- Use more than one visual affordance for actions like links.

Use Material's Accessible color system . This color system is based on tonal palettes, and is central to making color schemes accessible by default.

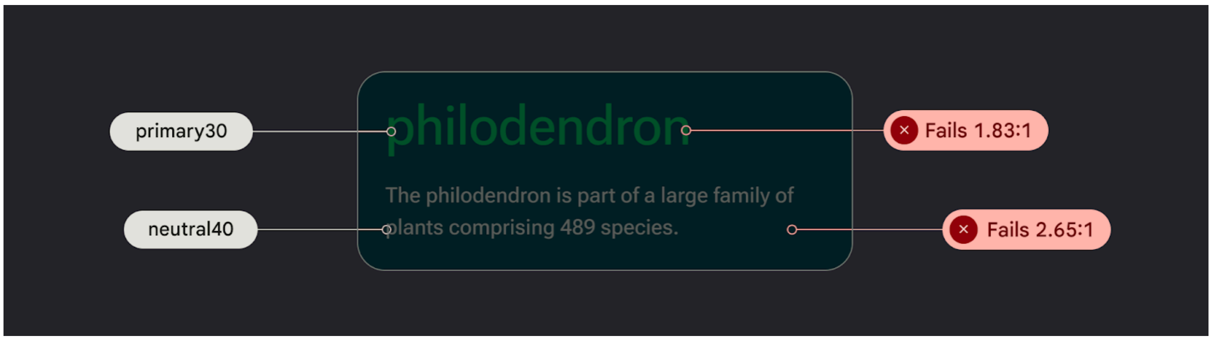

primary30
neutral40
philodendron
The philodendron is part of a large family of plants comprising 489 species.
Fails 1.83:1
Fails 2.65:1

Figure 1: *Example of text failing color contrast*

Design for sound

TalkBack is a Google screen reader included on Android devices that gives users eyes-free control. You can manually test this by exploring your app with TalkBack or with the A11y scanner.

Follow these guidelines to ensure your app is prepared for screen readers:

- Describe UI elements in your code. Compose uses Semantics properties to inform accessibility services about the information shown in UI elements.
- To satisfy Android framework requirements, provide additional textual description of icons and images.
- Set decorative item descriptions to null.
- To allow skipping between blocks of actions and content, consider UI granularity and group UI elements..

Check out Material's Design to Implementation Walk, which walks you through accessibility considerations and notation using Web Content Accessibility Guidelines (WCAG).

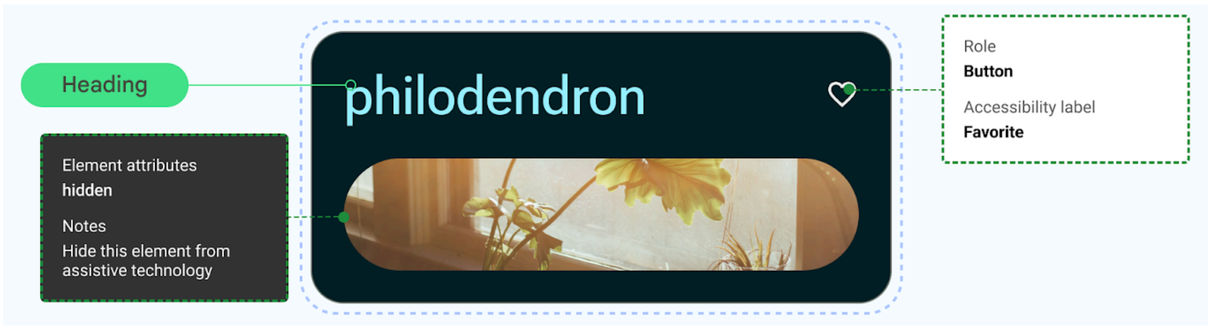


Figure 2: *UI elements labeled for accessibility: heading, hiding decorative image, and button label*

Design for audio

Android provides features to enable users to interact with their devices through a variety of voice commands and queries.

The Voice Access app for Android lets you control your device with spoken commands. Use your voice to open apps, navigate, and edit text hands-free.

Design for motor skill

Switch Access lets users interact with your Android device using one or more devices, which can be helpful for users with limited dexterity who have trouble interacting directly with a touch screen.

Manually test by exploring switch access.

- Don't rely on gestures to complete all actions; create accessibility actions to support all user flows in your app.
- Ensure all touch targets are at least 48 dp, even if this extends past the UI element visual.
- Consider haptic feedback to help inform the user with additional, real-time sensory input.

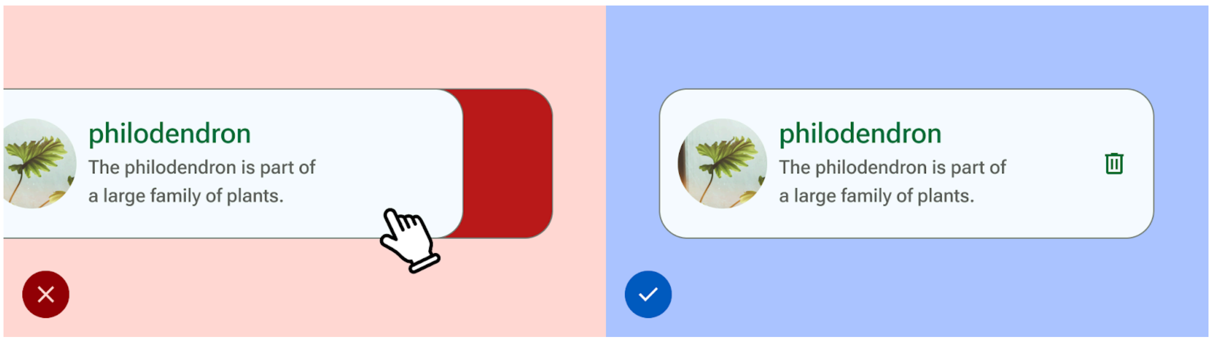

philodendron
The philodendron is part of a large family of plants.
philodendron
The philodendron is part of a large family of plants.

Figure 3: *The UI on the left lets the user delete only by swiping, while the UI on the right also provides an additional affordance in the form of a trash icon button.*

Android system bars
https://developer.android.com/design/ui/mobile/samples

bookmark_border

Together, the status bar and the navigation bar are called the *system bars*. They display important information such as battery level, the time, and notification alerts, and provide direct device interaction from anywhere.

It's critical to take the prominence of system bars into account, whether you're designing UI for interactions with the Android OS, input methods, or other device capabilities. Keep system bars at the top of most layers to ensure they're accounted for.

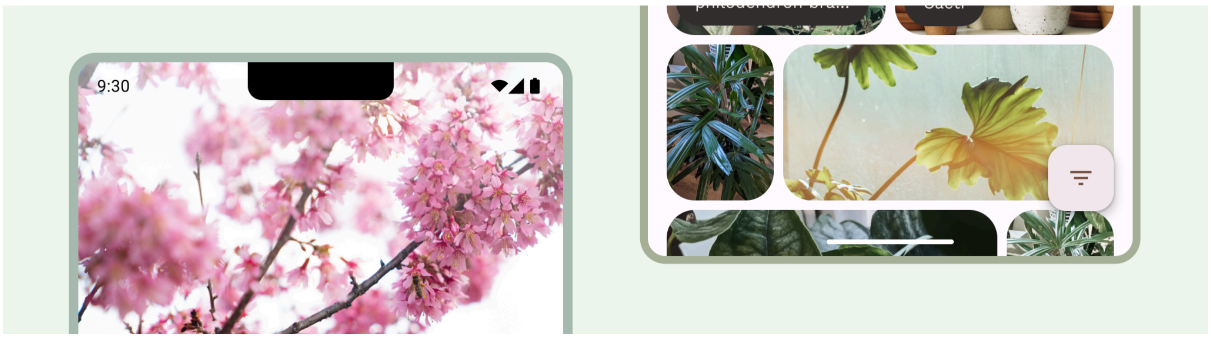


Figure 1: *Images behind system bars*

Takeaways

- Include system bars in your designs to account for UI safe zones, system interactions, input methods, display cutouts, and other device capabilities. Keep system bars at the topmost layer ensures they are accounted for.
- Do: Make system bars transparent and lay out your app in full screen, continuing the UI under the bars to give full edge-to-edge experience.

- If you can't set both bars to be transparent, ensure the colors of the bars match the color of the body of your app. For example, match the bottom navigation bar color with the gesture bar color, and the top status bar color with the body color.

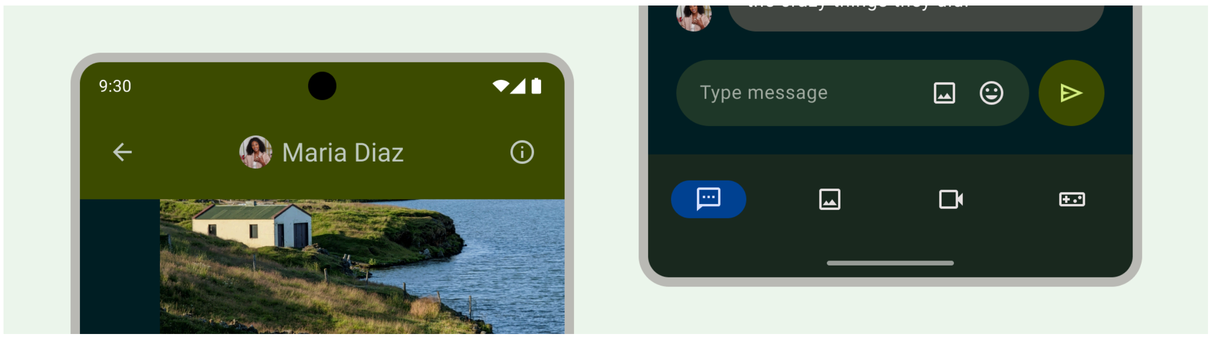


Figure 2: *Ensure the system bar colors match the body color of your app*

- Avoid adding tap gestures or drag targets under gesture insets; these conflict with edge-to-edge and gesture navigation.

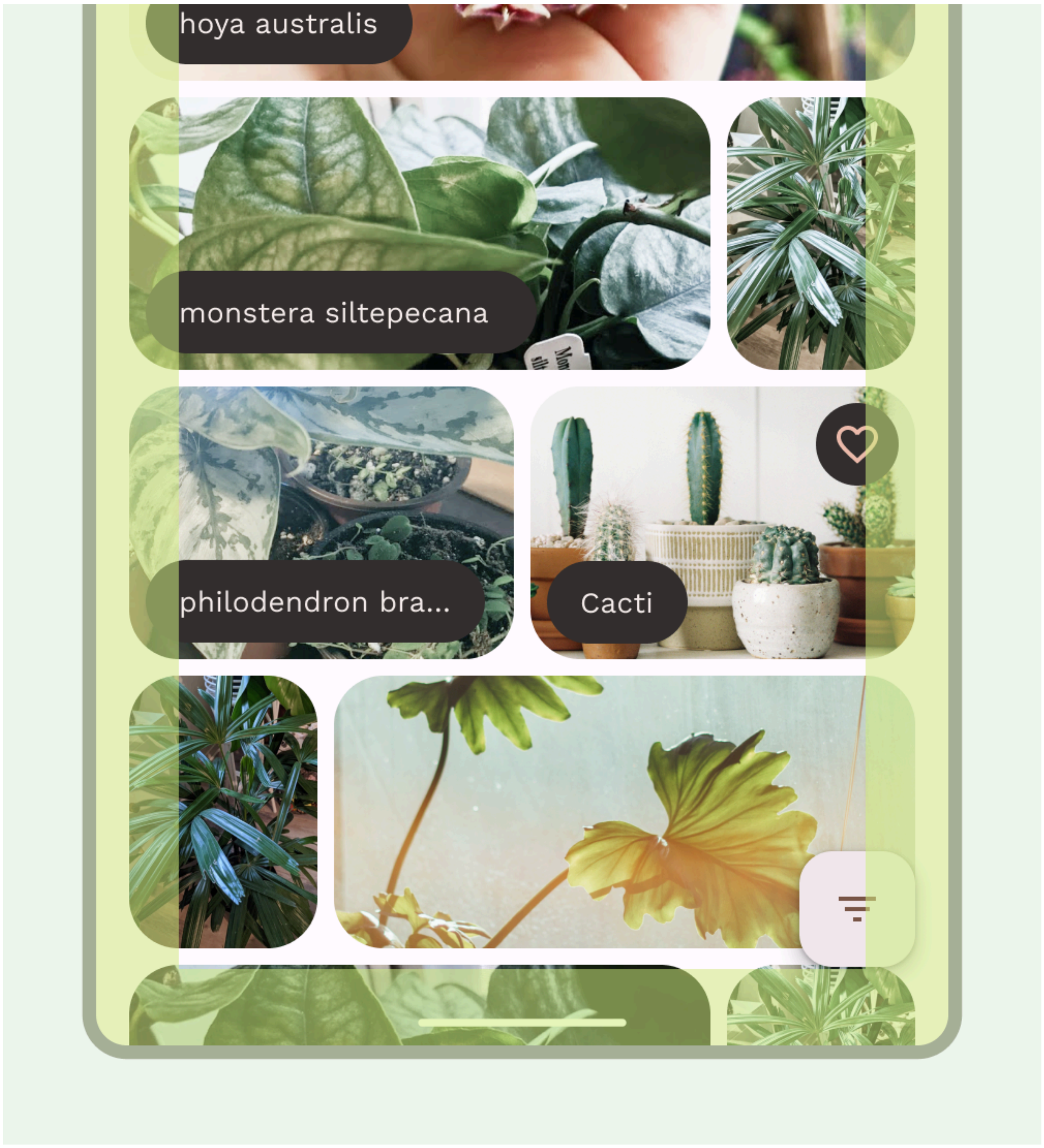


Figure 3: *System gesture insets. Avoid placing tap targets under these areas***Note:** System bars and navigation can vary by device manufacturer. Older devices may have hardware navigation buttons instead of software buttons.

Draw your content behind the system bars

The edge-to-edge feature allows Android to draw the UI under the system bars for a more immersive experience. We recommend using edge-to-edge because making the navigation bar transparent is a common request from users. (Read about how to support edge-to-edge).

Note: Don't hide the system bars, except temporarily for immersive mode experiences such as watching a full-screen video.

An app can address overlaps in content by reacting to *insets*. Insets describe how much the content of your app needs to be padded to avoid overlapping with parts of the Android OS UI such as the navigation bar or status bar, or with physical device features such as display cutouts.

Be aware of the following types of insets when designing for edge-to-edge use cases:

- *System bars insets* apply to UI that is both tappable and that shouldn't be visually obscured by the system bars.
- *System gesture insets* apply to gesture-navigational areas used by the system that take priority over your app.

Status bar

On Android, the status bar contains notification icons and system icons. The user interacts with the status bar by pulling it down to access the notification shade.

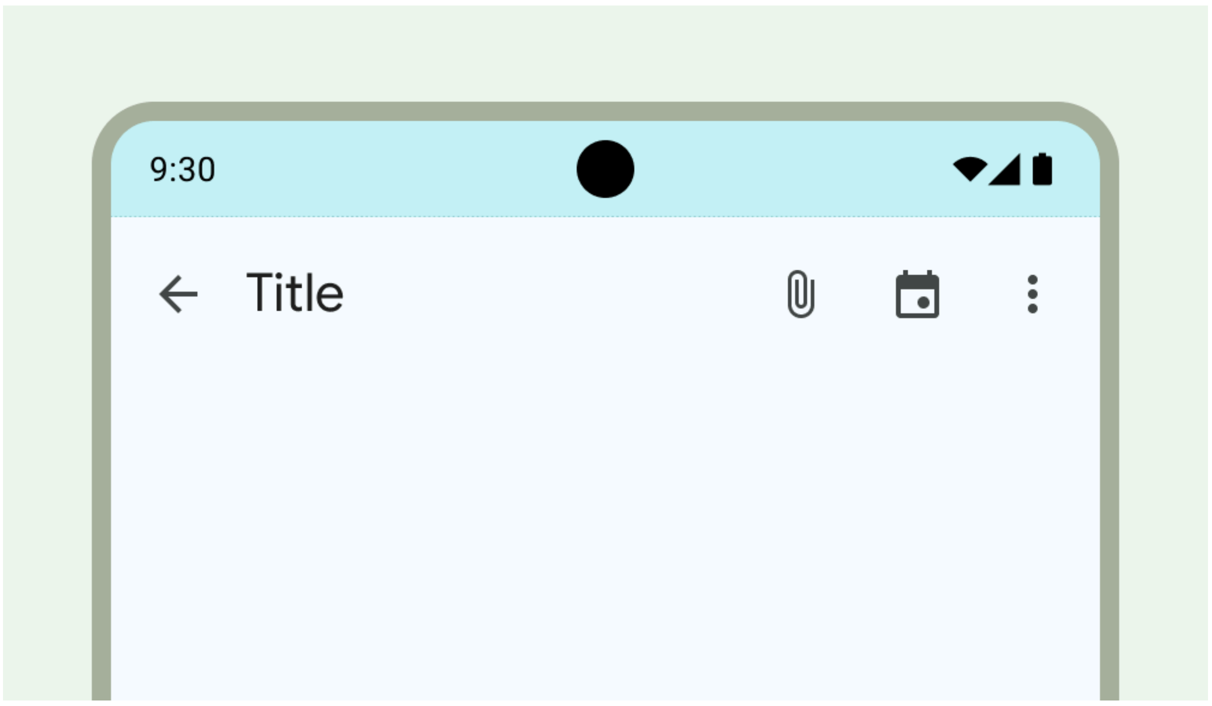


Figure 4: *Status bar region highlighted on top of top app bar*

The status bar can appear differently depending on the context, time of day, user-set preferences or themes, and other parameters. You can also set the status bar style, as in the following examples.

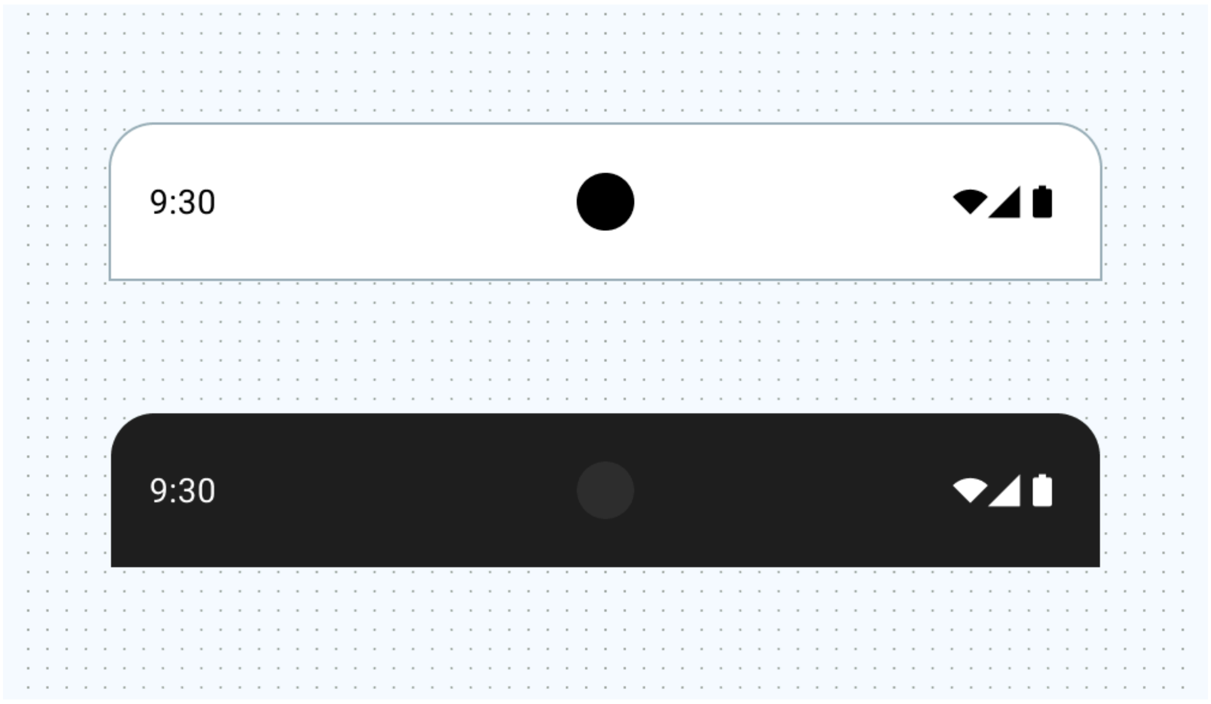


Figure 5: *Status bar in light and dark theme.*

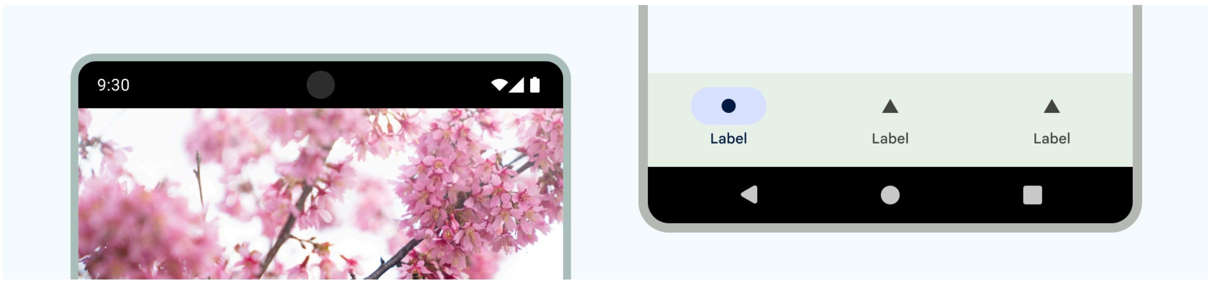


Figure 6: *Default bars (black)*

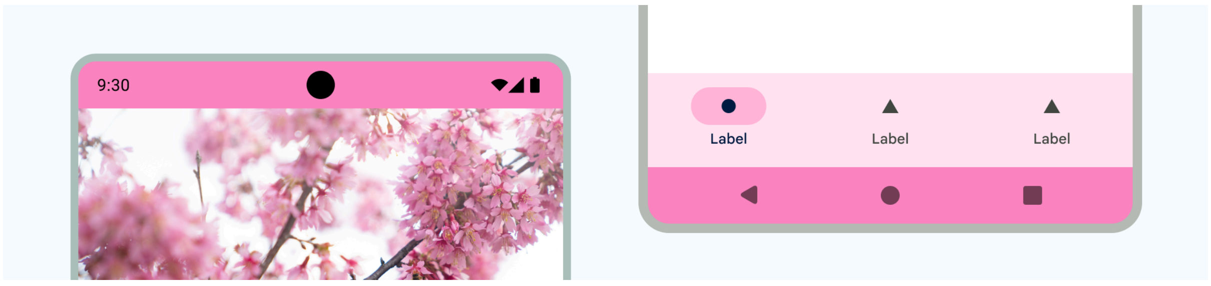


Figure 7: *Styled bars*

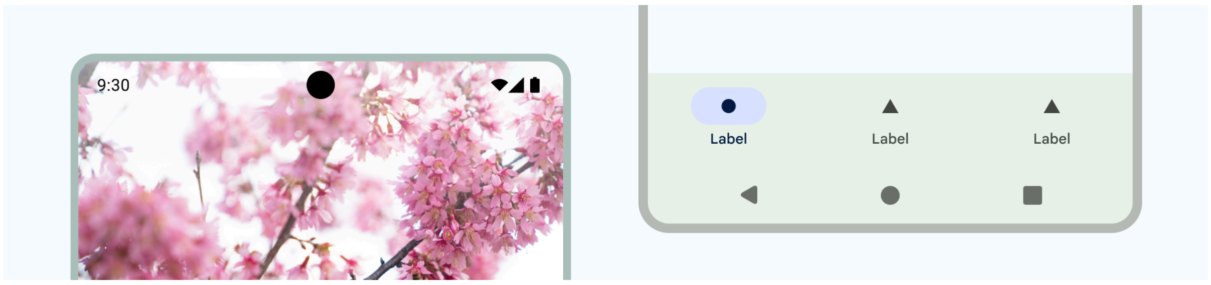


Figure 8: *Transparent bars using the [edge-to-edge feature](), ideal for letting your content shine through using the most screen space.*

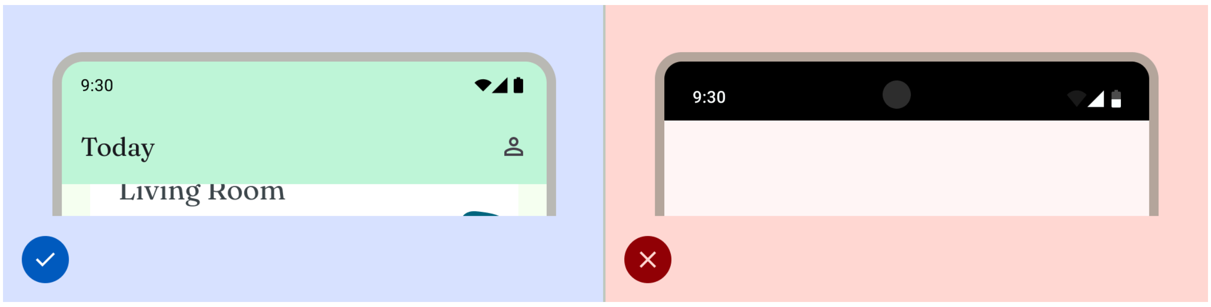


Figure 9: *Do style your system bars to enhance your content or match your app's branding. Don't leave the system bars set to the default attributes.*

When a notification arrives, an icon usually appears in the status bar. This signals to the user that there's something to see in the notification drawer. This can be your app icon or symbol to represent the channel. See [Notifications design]().

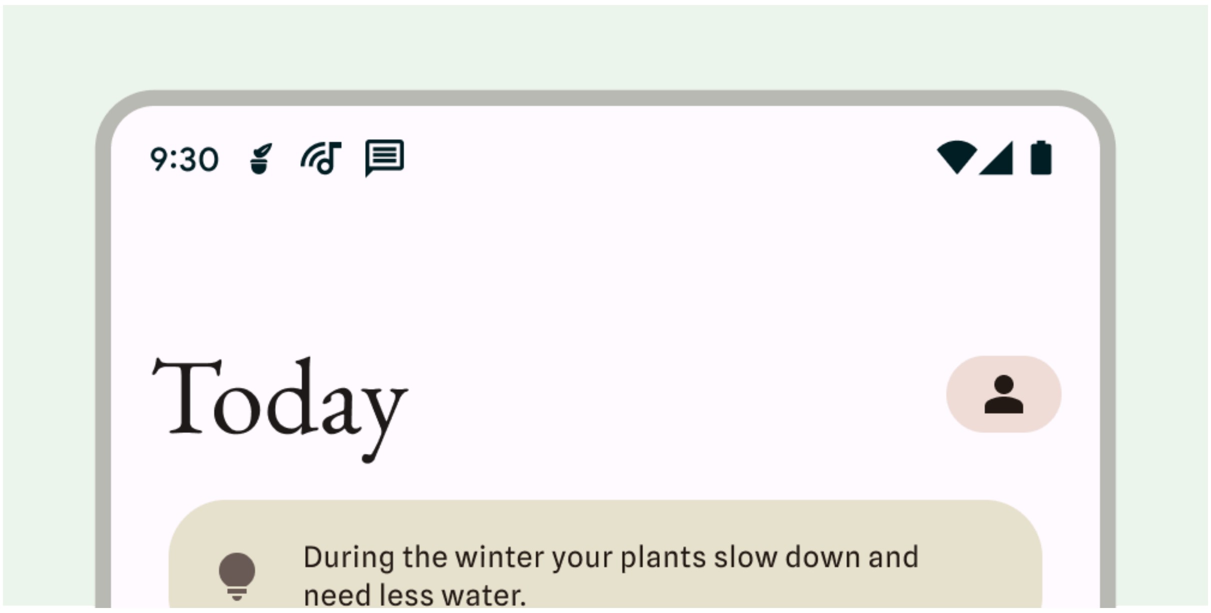

9:30
Today
During the winter your plants slow down and need less water.

Figure 10: *Notification icon in the status bar*

Set the status bar style

Style the status bar background as a part of your app theme, with a custom color or style, along with setting transparency and opacity.

```
<style name="Theme.MyApp">
  <item name="android:statusBarColor">
    @android:color/transparent
  </item>
</style>
```

If you're manually setting the background color, you can optionally style status bar contents as light or dark for optimal contrast.

Display cutouts and the status bar

A display cutout is an area on some devices that extends into the display surface to provide space for front-facing sensors. It can affect the appearance of status bars. Display cutouts can vary depending on the manufacturer.

Read about how to support display cutouts.

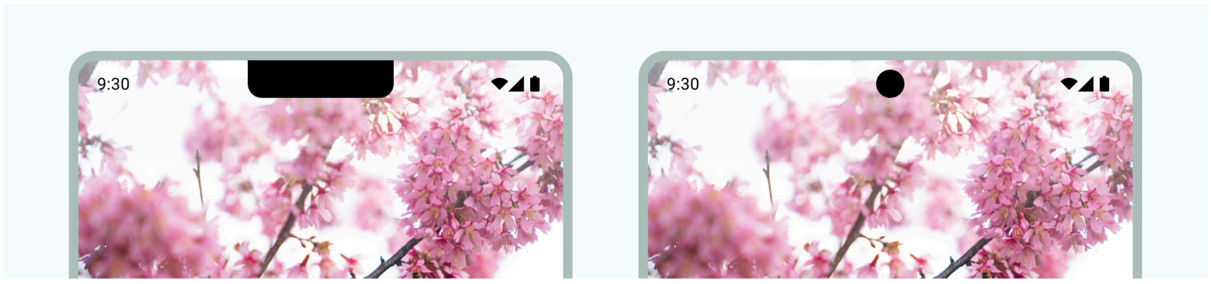


Figure 11: *Examples of display cutouts*

Navigation bar

Android lets users control navigation using back, home, and overview controls:

- Back returns directly back to the previous view.
- Home transitions out of the app and to the device's home screen.
- Overview shows the apps are open and have recently been opened.

Users can choose from various navigation bar configurations including gesture navigation (recommended) and three-button navigation.

Gesture navigation

Introduced in Android 10 (API level 29), gesture navigation is the recommended type of navigation, although you can't override the user's preference. Gesture navigation doesn't use buttons for back, home, and overview, instead showing a single gesture handle for affordance. Users interact by swiping from the left or right edge of the screen to go back and forward and up from the bottom to go home. Swiping up and holding opens the overview.

Gesture navigation is a more scalable navigation pattern for designing across mobile and larger screens. To provide the best user experience, account for gesture navigation by:

- Supporting edge-to-edge content.
- Avoid adding interactions or touch targets under gesture navigation insets.

Read about implementing gesture navigation.

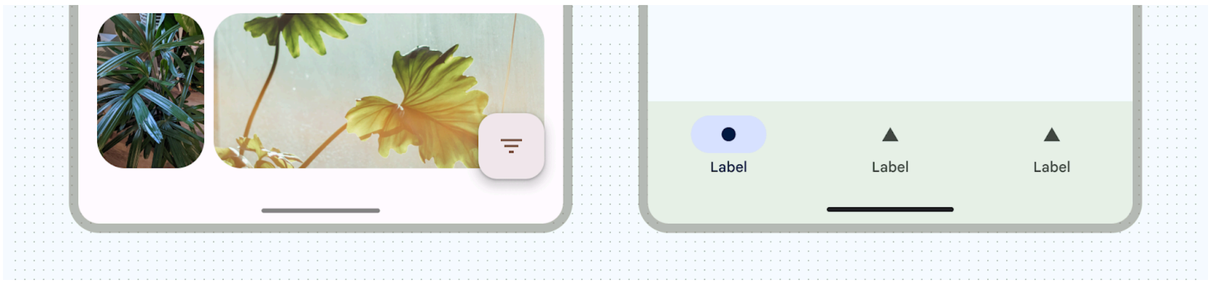


Figure 12:** Gesture handle navigation barTip:** There's no way to determine which type of navigation the user has enabled. To deliver the best experience, account for multiple types of navigation. If you're in the position of having to prioritize your work, it's best to prioritize gesture navigation because it's the type of navigation many existing and new Android users are most familiar with.

Three-button navigation

Three-button navigation provides three buttons for back, home, and overview.

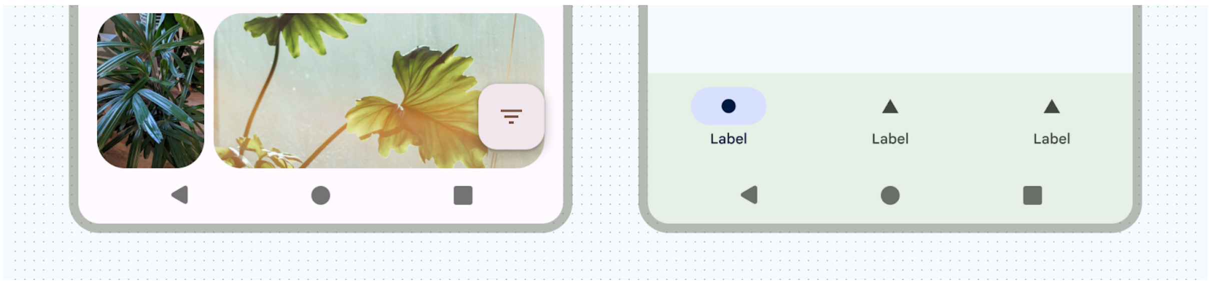


Figure 13: *Three-button navigation bar*

Other navigation bar variations

Depending on Android version and device other navigation bar configurations may be available to your users. Two-button navigation, for example, provides two buttons for home and back.

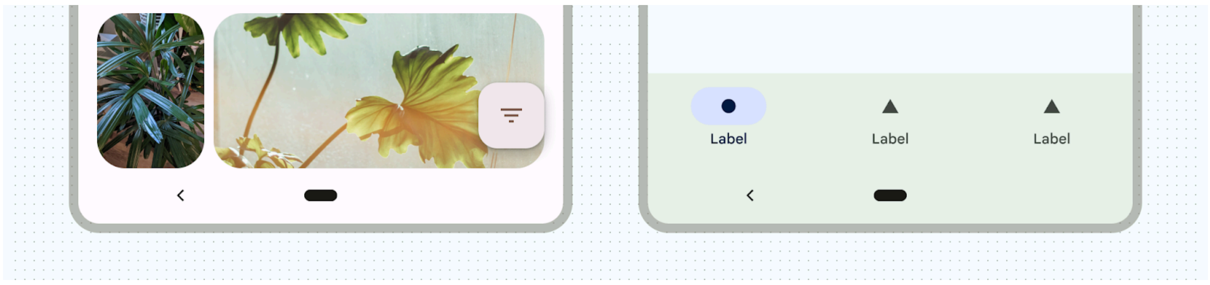


Figure 14: *Two-button navigation bar*

Set a navigation style

The following example shows how to implement a navigation style.

```
<style name="Theme.MyApp">

  <item name="android:navigationBarColor">

    @android:color/transparent

  </item>

</style>
```

Android handles all visual protection of the user interface in gesture navigation mode or in the button modes.

- **Gesture navigation mode**: The system applies dynamic color adaptation, in which the contents of the system bars change color based on the content behind them. In the following example, the handle in the navigation bar changes to a dark color if it's placed above light content, and vice versa.

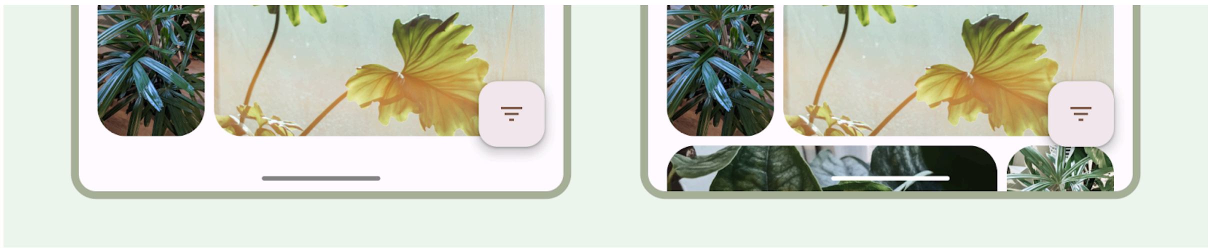


Figure 15: *Dynamic color adaptation*

- **Button modes**: The system applies a translucent scrim behind the system bars (for API level 29 or higher) or a transparent system bar (for API level 28 or lower).

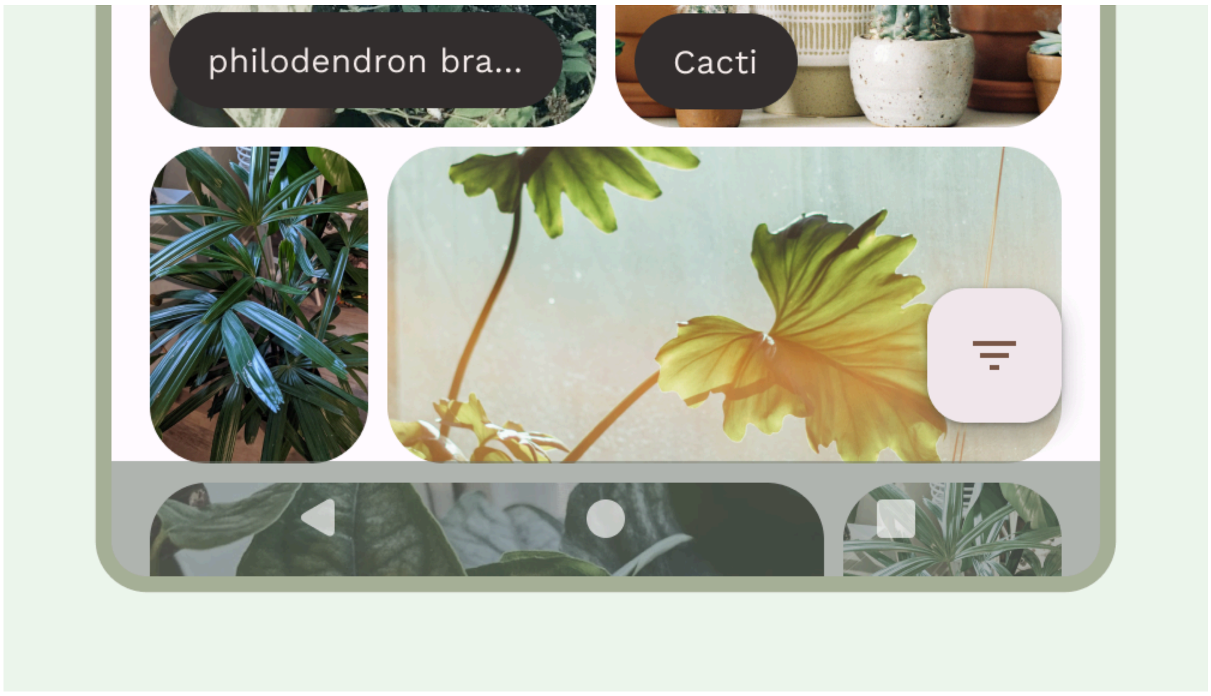


Figure 16: *Dynamic color adaptation, where system bars change color based on the content behind them*

Keyboard and navigation

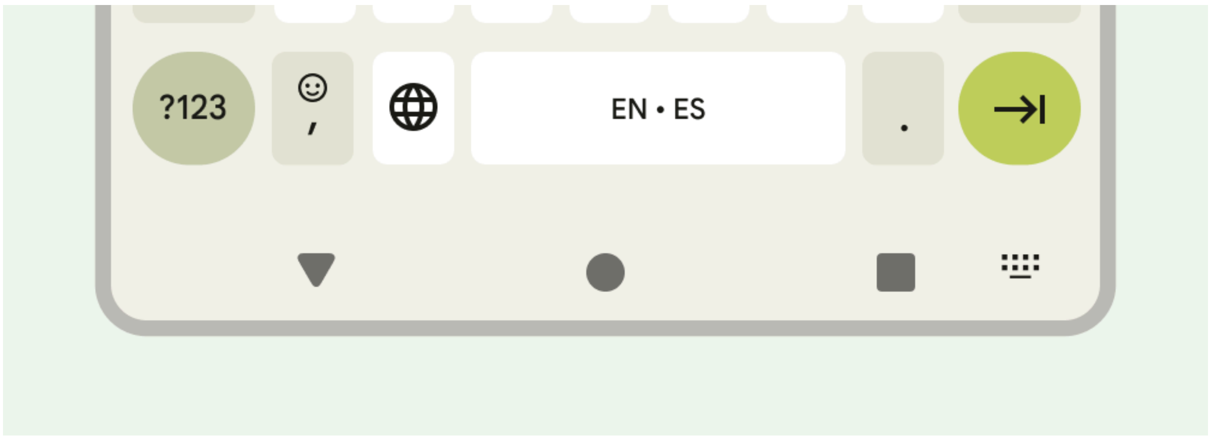


Figure 17: *On-screen keyboard with navigation bars*

Each navigation type reacts appropriately to the on-screen keyboard to allow the user to perform actions such as dismissing or even changing the keyboard type. To ensure a smooth keyboard transition, To ensure a smooth transition that synchronizes the transition of the app with the keyboard sliding up and down from the bottom of the screen, use `WindowInsetsAnimationCompat`.

Immersive mode

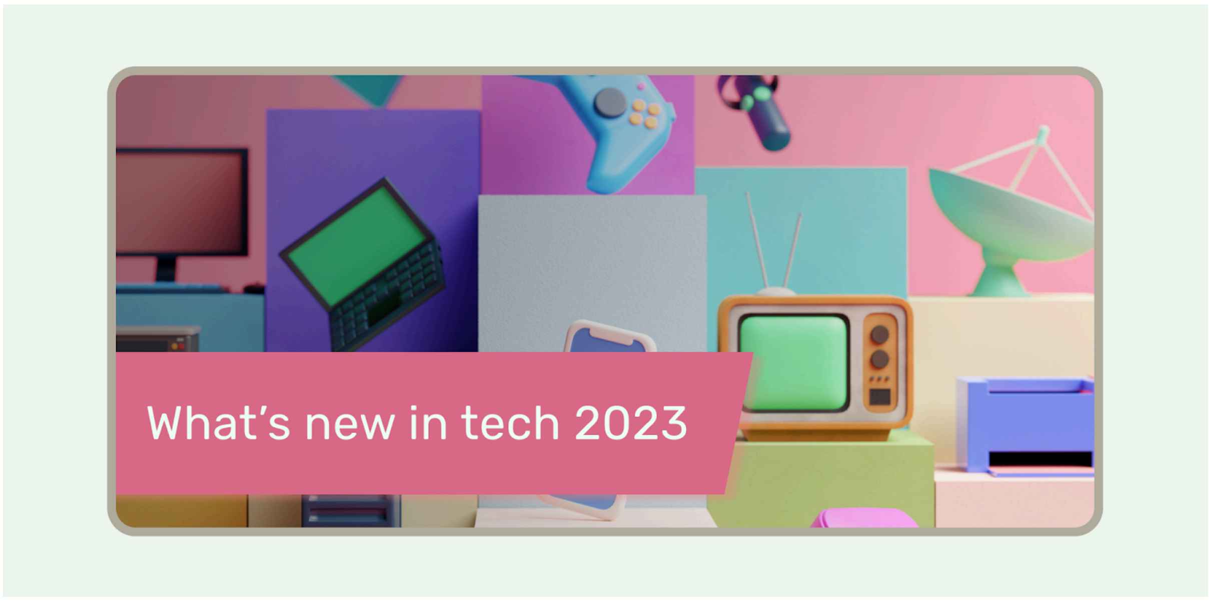
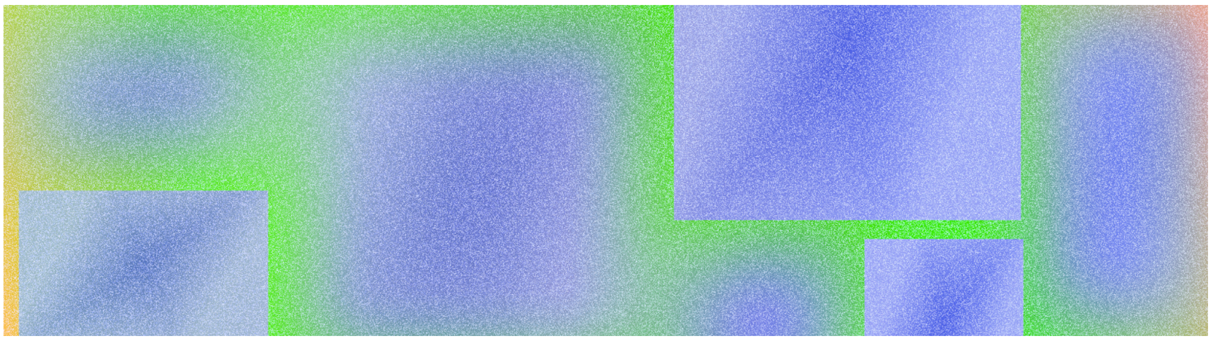


Figure 18: *Immersive mode showing a full-screen experience on a landscape-oriented mobile device*

You can hide system bars when you need a full-screen experience, for example when the user is watching a movie. The user should still be able to tap to reveal system bars and UI in order to navigate or interact with system controls. Learn more about designing for full screen modes, or read about how to hide the system bars for immersive mode.

Layout basics

bookmark_border



A layout defines the visual structure for a user to interface with your app, such as in an activity. Android provides a range of libraries, canonical starting points, and techniques to display and position content.

Takeaways

- Honor device safe areas, which includes parts of the UI such as display cutouts, edge-to-edge insets, edge displays, software keyboards, and system bars.
- **Do:** Provide a flexible layout for users to interact with the keyboard.
  Video 1: *Providing a flexible layout for users to interact*

Warning: Be careful when covering content with the keyboard.

- Keep essential interactions, like primary navigation, in a reachable screen area.

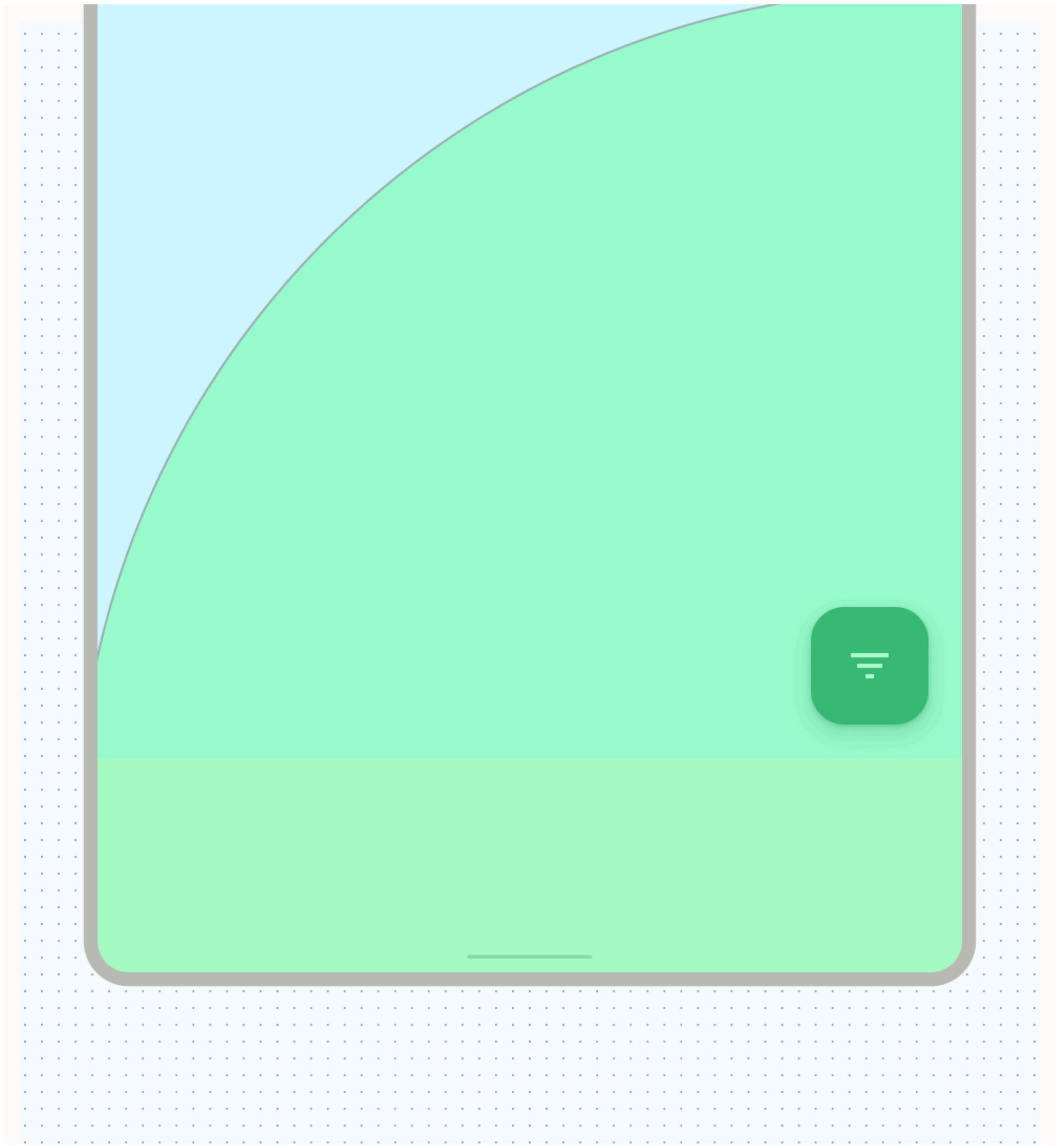


Figure 1: *Floating action buttons (FABs) provide a prominent and reachable interaction point*

- Use containment to group related content to guide the user through content and actions.

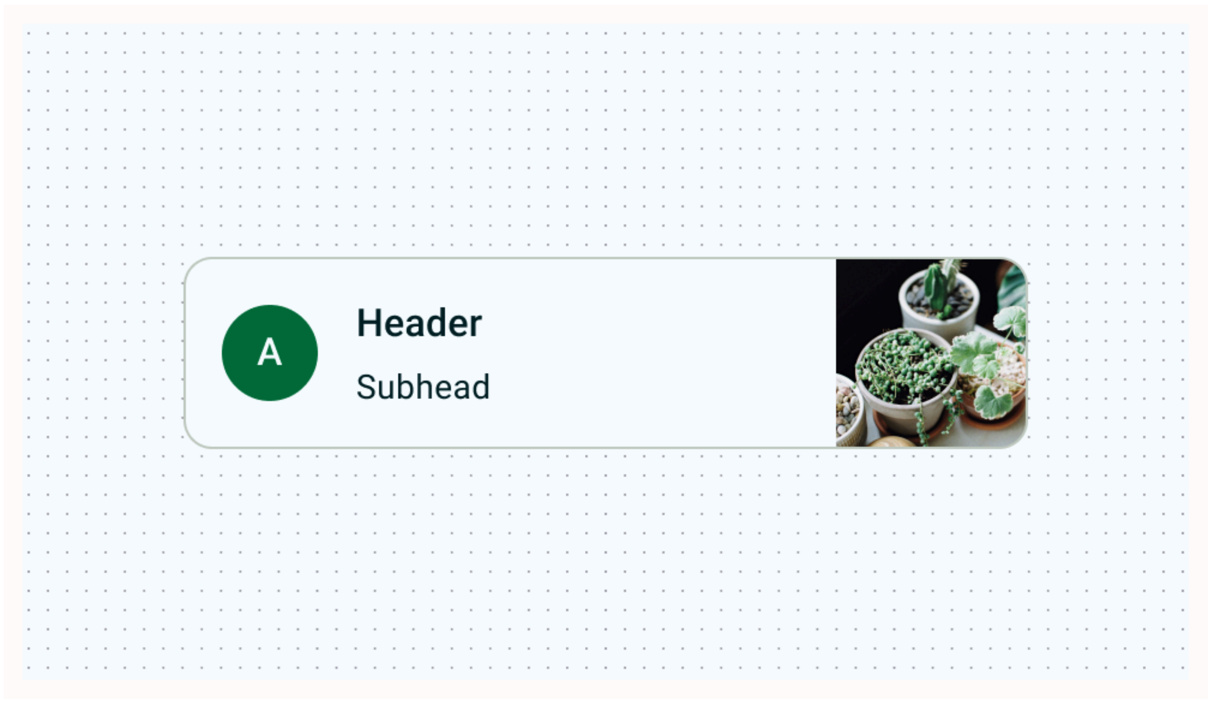


***Figure 2:** Cards using explicit containment to group content with related actions*

- Provide consistent alignment between similar content and UI elements.
  Don't: disrupt readability by inconsistently spacing like elements, which can make designs appear haphazard.
  Do: Establish consistent spacing between like elements.

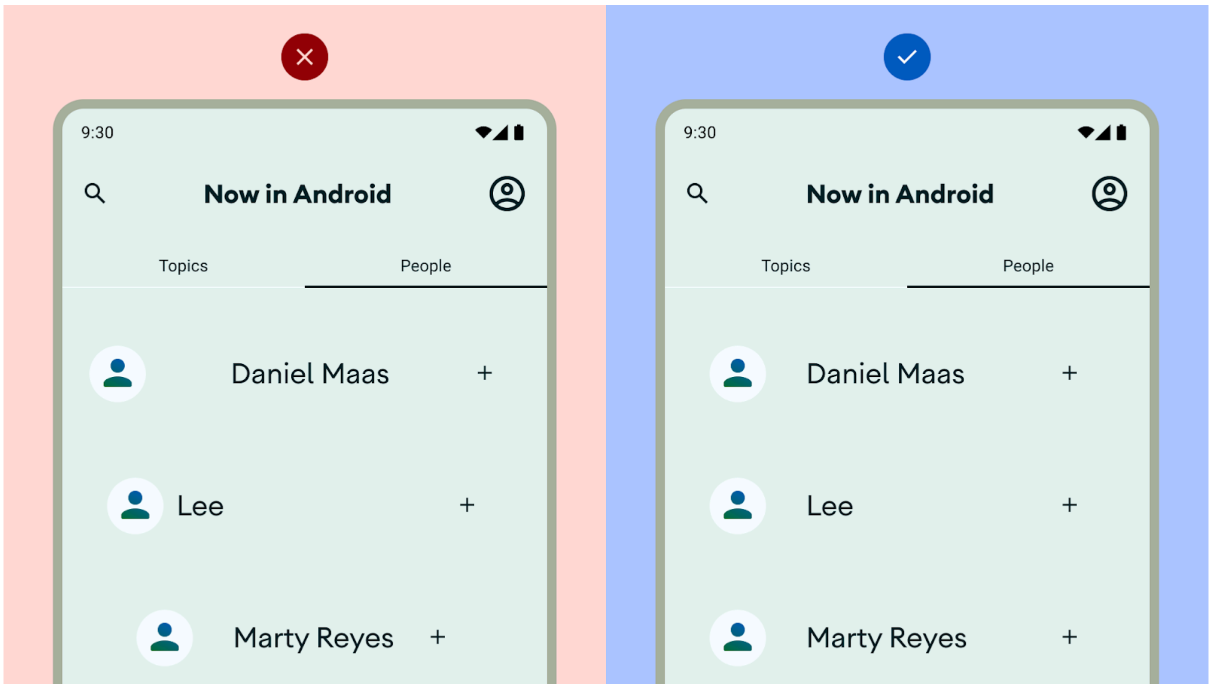


***Figure 3:** Don't disrupt readability*

- Don't stick to portrait or an idealized layout: Consider different aspect ratios, size classes, and resolutions that users may encounter.
- Don't overwhelm your user with too many actions per view.

- When building custom layouts, notate how content should sit within the layout using alignment, constraints, or gravity terms. Include how images should respond to their container to display properly.

Parts of a typical Android app screen

Most Android apps consist of regions referred to as the system bars, the navigation area, and the body.

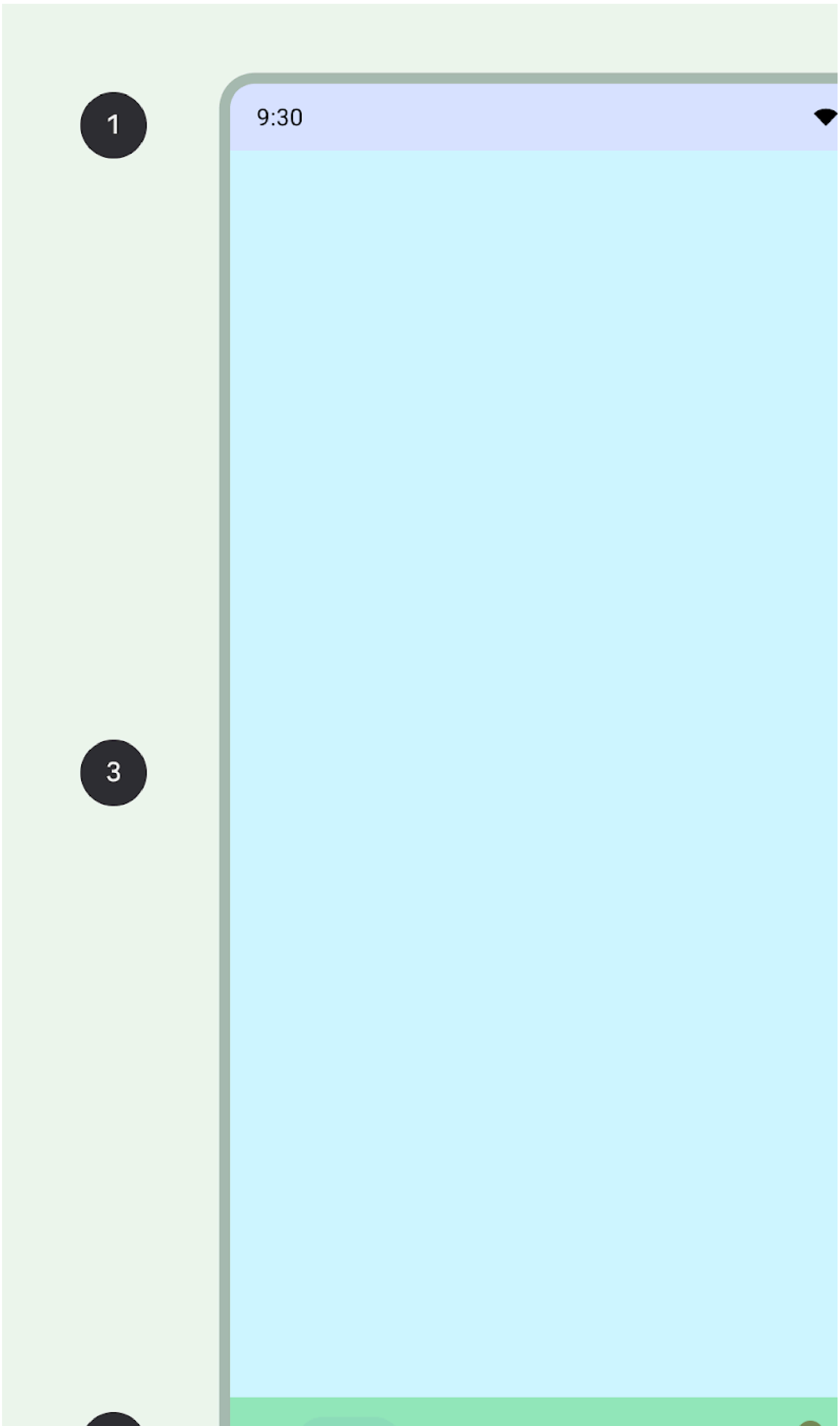

1
9:30
3

Figure 4: *Parts of an Android app: systems bars ( ), navigation area ( ), and body ( )*

System bars

The status bar and navigation bar–collectively known as the system bars–display important information such as battery level, the time, and notification alerts, and provide direct device interaction from anywhere. Read more about system bars.

System bars are an integral part of the device interface. Add them as a top layer of your designs to ensure they're accounted for within the screen layout. Otherwise, users might mistakenly assume the intent is to hide them, you miss out on styling them, and spacing can end up being off.

Add the bars as a top layer. Use `android:navigationBarColor` and `android:statusBarColor` to apply colors to the system bars in your app's theme.

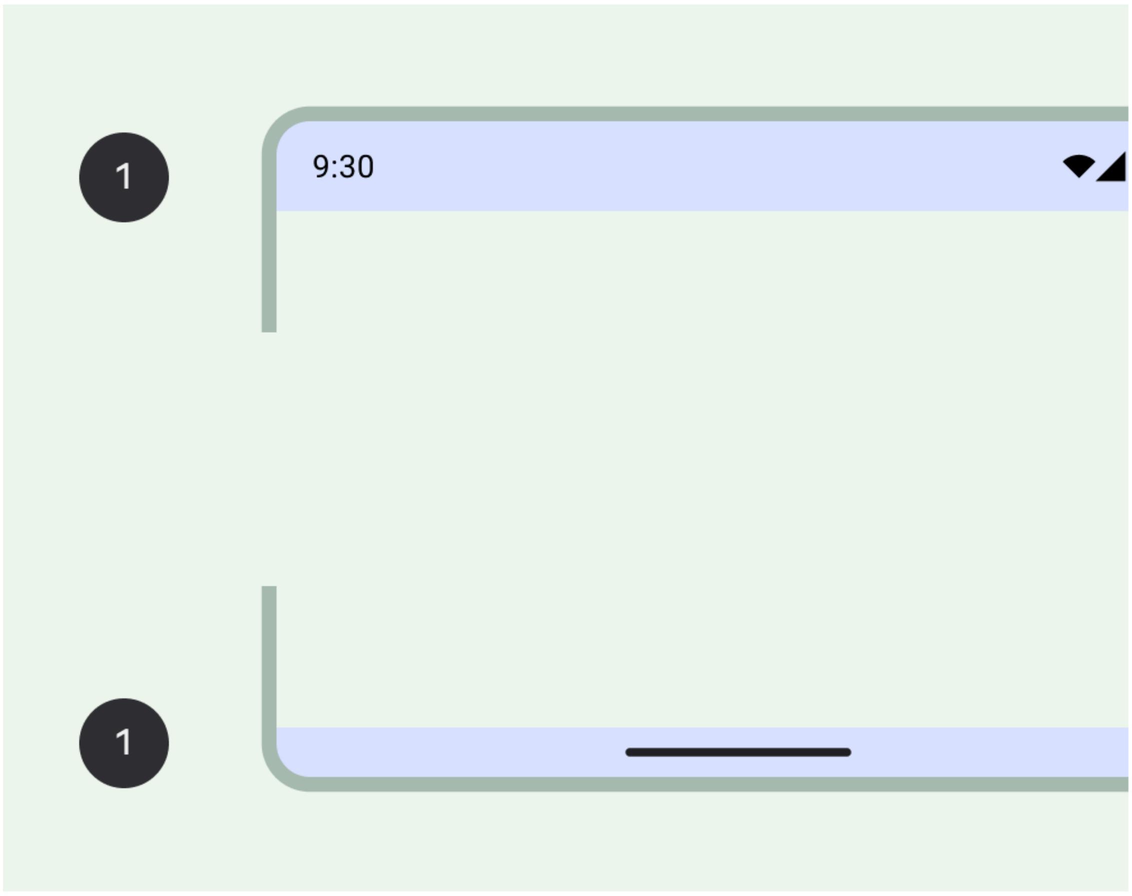


Figure 5: *System bars ( )*

Navigation region

Navigation represents the different affordances that allow a user to navigate within your app, access important actions, or across the Android platform.

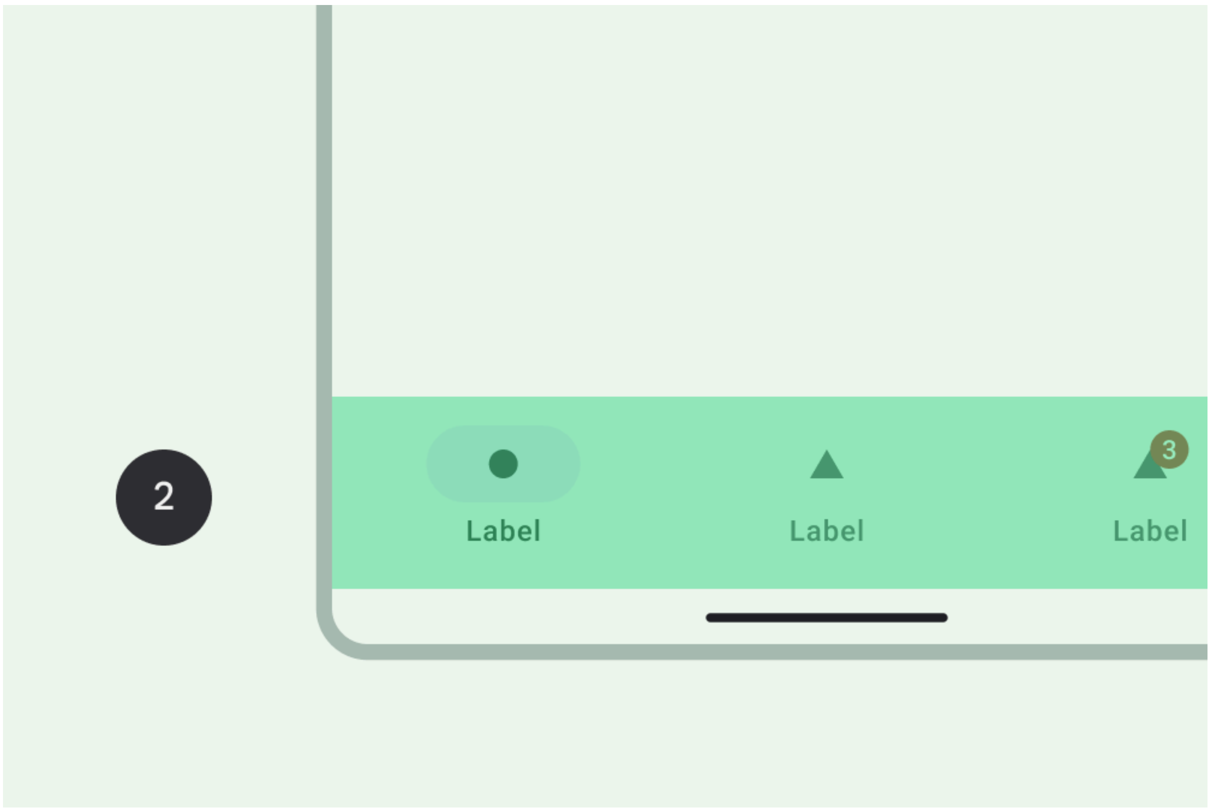


Figure 6: *Navigation area ( )*

Body region

The body region holds the screen content. Body content is composed of additional groupings and layout parameters. It must continue under navigation and system bar regions.

Declare `WindowCompat.setDecorFitsSystemWindows(window, false)` for edge-to-edge insets.

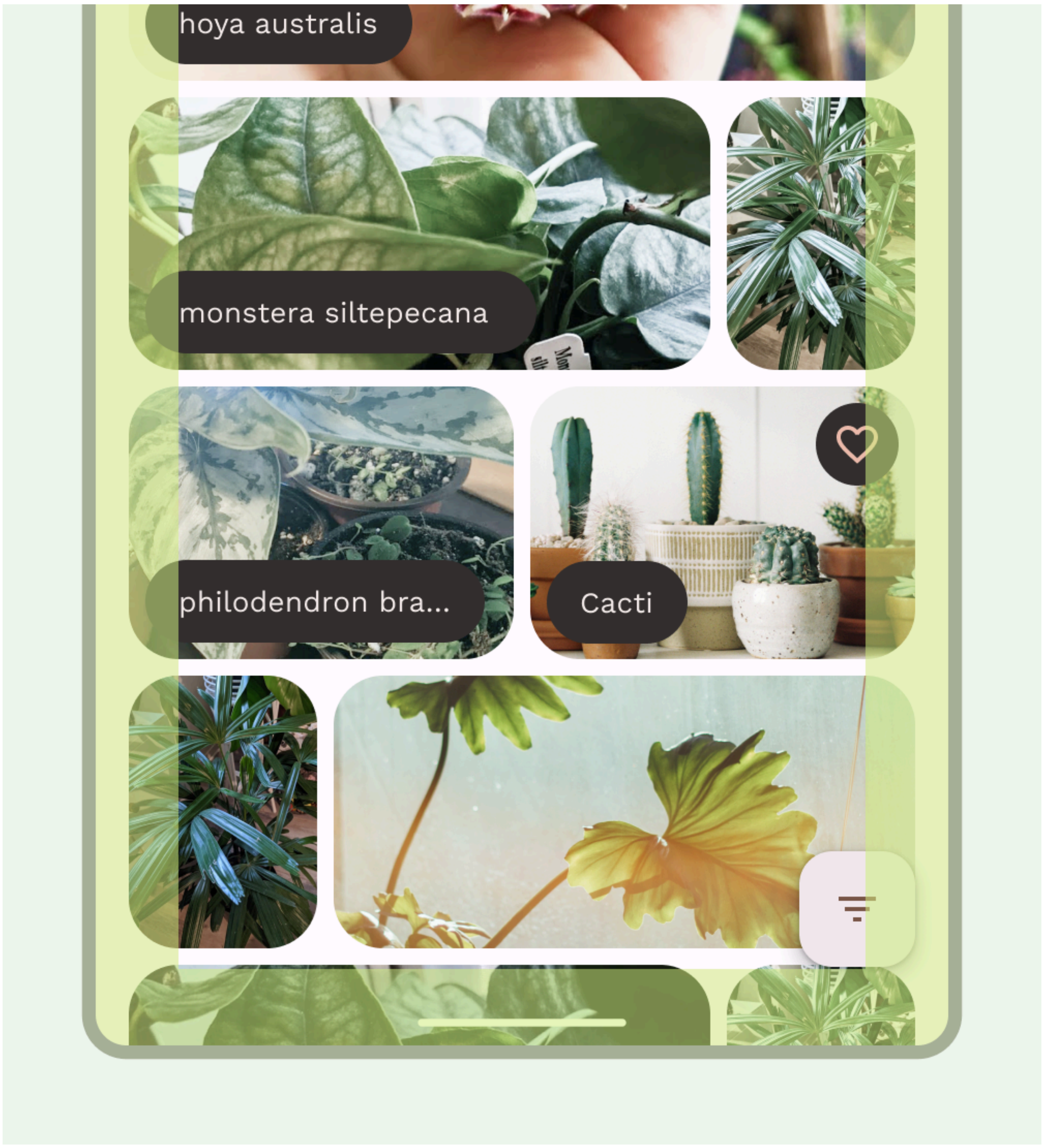


Figure 7: *Body region*

To determine the appropriate composition and navigation patterns for your layout, seek to understand how users interact with your content, and how they navigate your app's information architecture. This understanding can guide your design toward being more user-focused by creating UI that users can act on.

Content composition and structure

Build up a flexible flow and rhythm with a structure and containment methods for your content.

Base structure: use margins and columns for visual guardrails

To begin creating a solid structure with consistent guardrails, add margins and columns to your layouts.

Margins provide spacing on the left and right edges of the screen and content. **A standard margin value for compact sizing is 16 dp, but margins should adapt to accommodate larger screens. Your app's body content and actions must stay within and align with these margins.**

You can also ensure inset safe zones or insets at this step. System bar insets prevent crucial actions from falling under system bars. See [Draw your content behind the system bars]() for details.

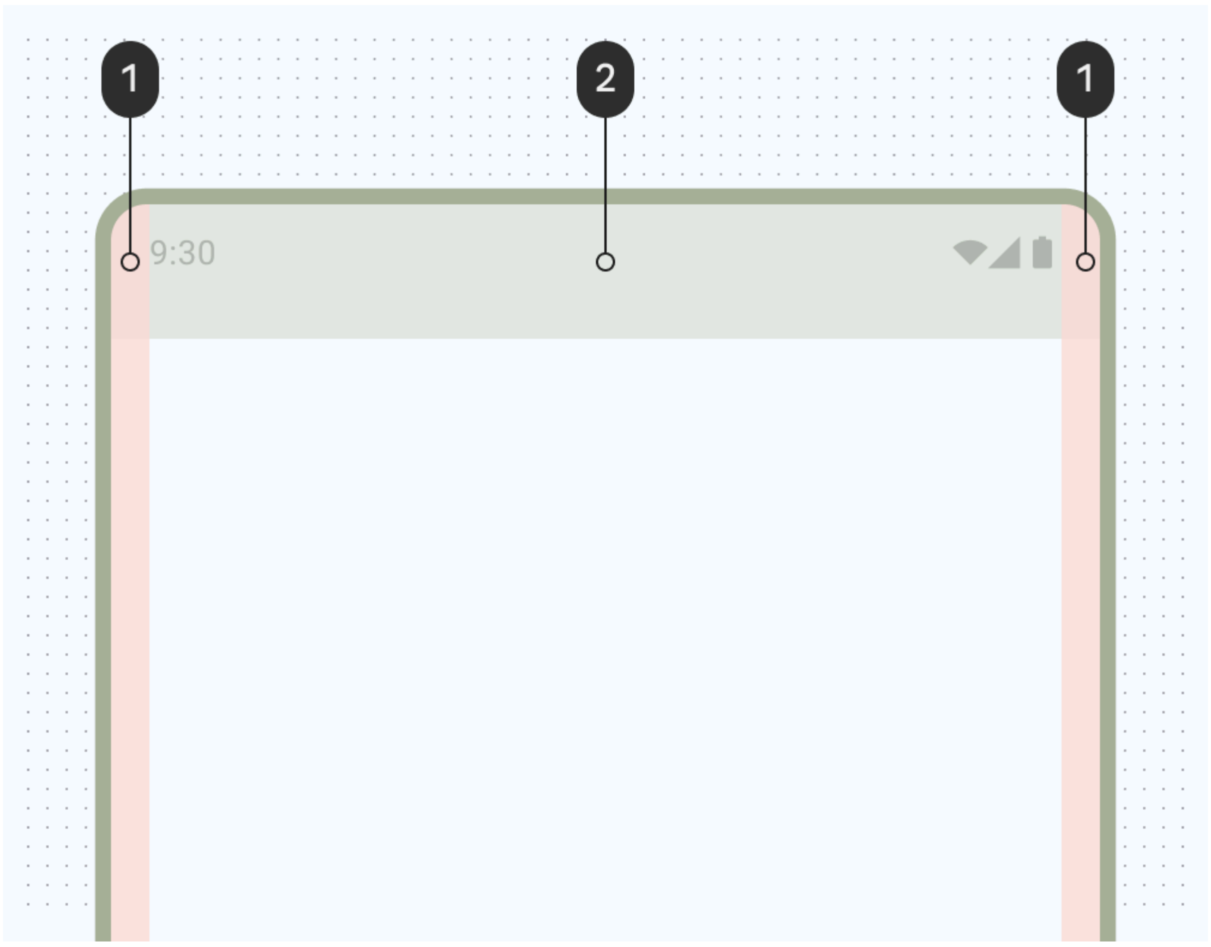


Figure 8: *Margins ( ) and system bar inset ( )*

Use *columns* to build a flexible grid structure for consistent alignment, and to provide vertical definition to a layout by dividing content within the body area. Content goes in the areas of the screen containing columns. These columns lend structure to your layout, providing convenient structure to arrange elements.

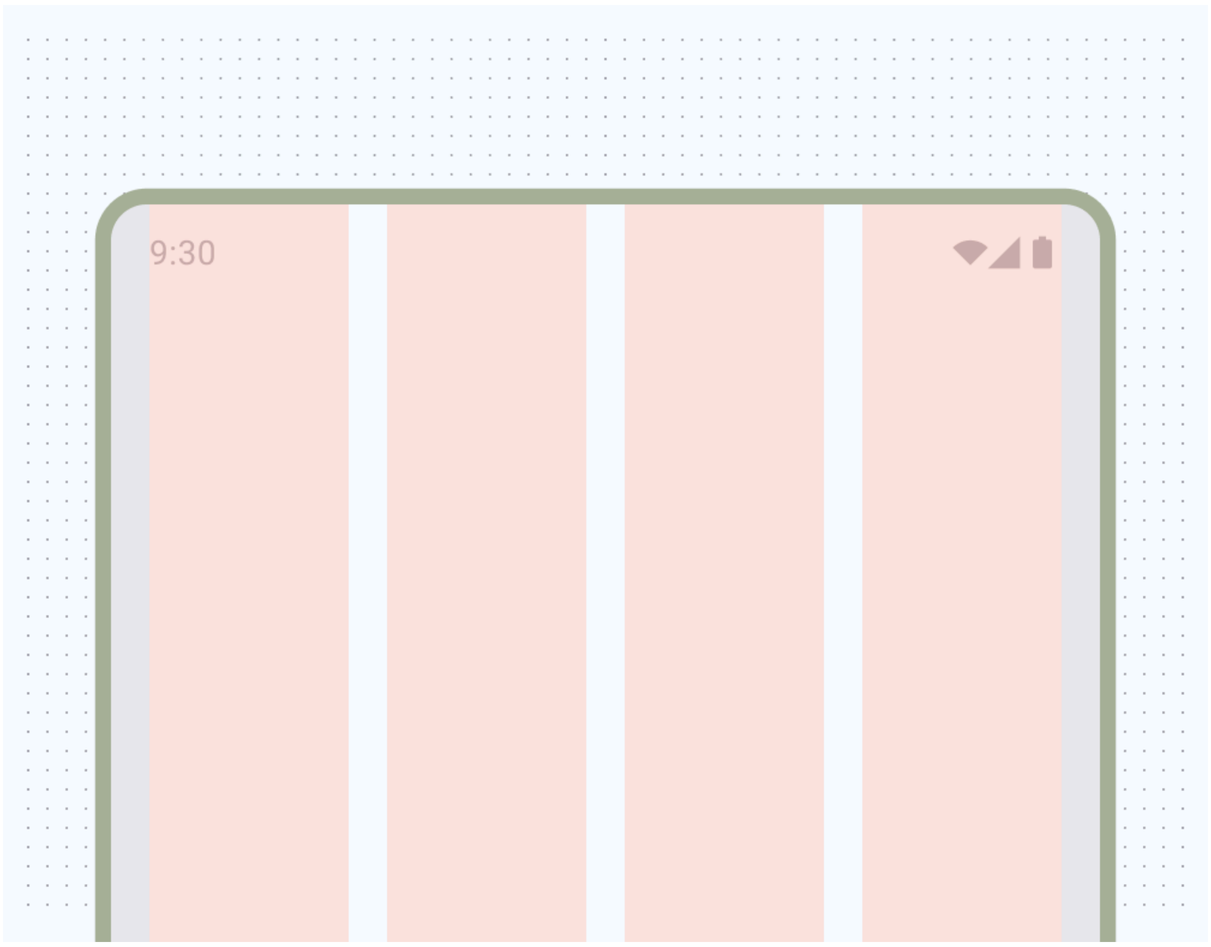


Figure 9: *Mobile screens are often divided into four columns*

Use a *column grid* to align content with an underlying grid but retain flexible sizing. The column grid can accommodate different form factors by changing the column sizes and number of columns as needed by the screen size at certain points while allowing content to also scale. Avoid being too granular with the column grid, this is what the baseline grid is for: providing consistent spacing units.

Be careful of establishing an accompanying grid of rows as it can restrict horizontal content scaling across orientations and form factors, typically establishing padding rules will provide the needed visual consistency.

Video 2: *Starting to add copy to the layout structure. Margins protect content from the screen's edges. Columns provide a consistent spacing and alignment structure.*

Use containment to visually group elements

Containment refers to using white space and visible elements together to create a visual grouping. A container is a shape that represents an enclosed area. In a single layout, you can group together elements that share similar content or functionality and separate them from other elements using open space, typography, and dividers.

You can group similar items together with white space or visible division to help guide the user through content.

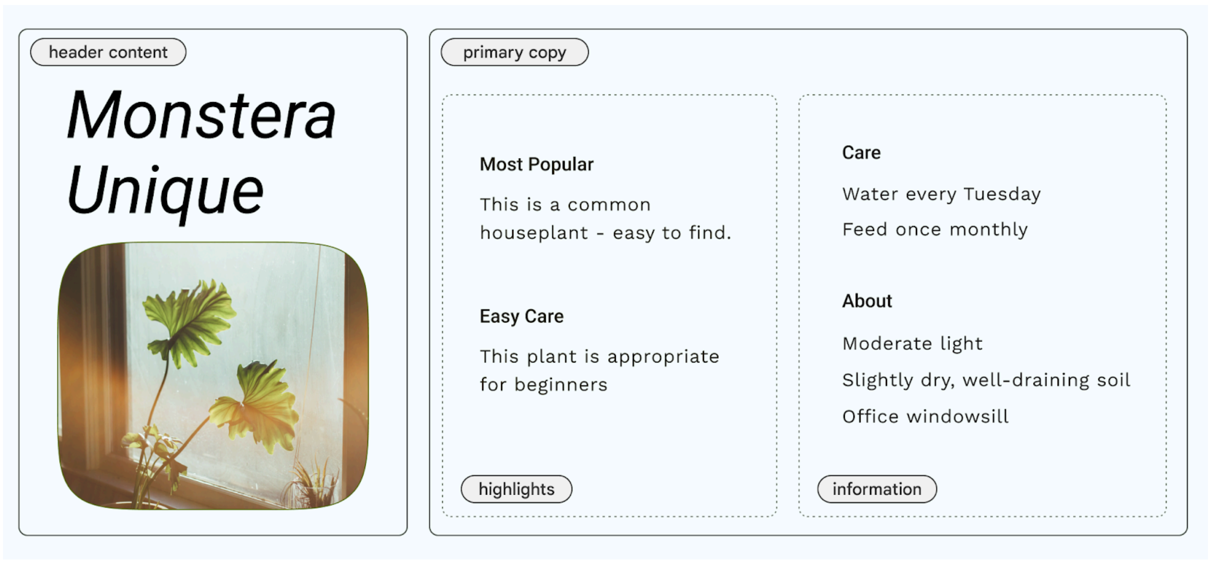


Figure 10: *Breaking content into two larger groupings of header and primary copy*

Implicit containment uses white space to visually group content to create container boundaries while *explicit containment* uses objects like divider lines and cards to group content together.

The following figure shows an example of using implicit containment to contain the header and primary copy. The column grid is used to align and create groupings. Highlights are explicitly contained within cards. Using iconography and type hierarchy for greater visual separation.

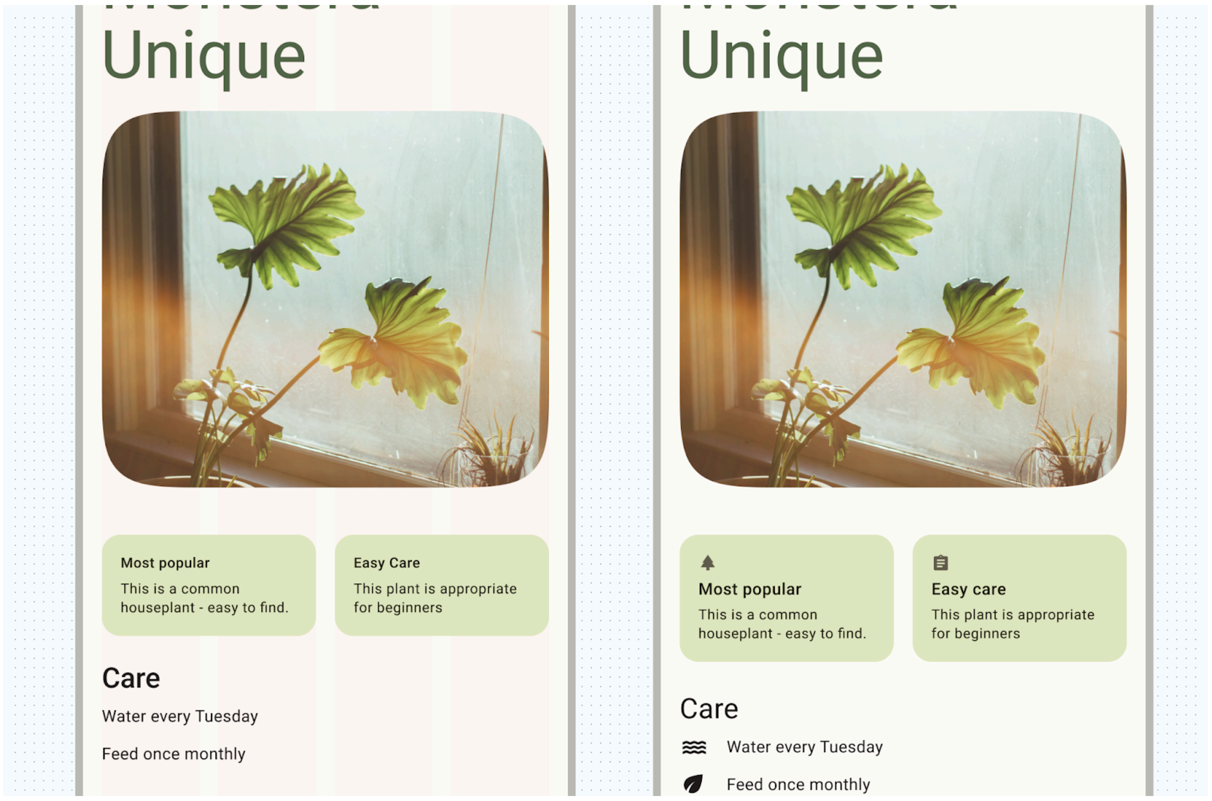

Unique
Most popular
This is a common houseplant - easy to find.
Easy Care
This plant is appropriate for beginners
Care
Water every Tuesday
Feed once monthly
Unique
Most popular
This is a common houseplant - easy to find.
Easy care
This plant is appropriate for beginners
Care
Water every Tuesday
Feed once monthly

Figure 11: *Example of implicit containment*

Positioning of content

Android has multiple ways to handle content elements in their respective containers that can help position them appropriately, including gravity, spacing, and scaling.

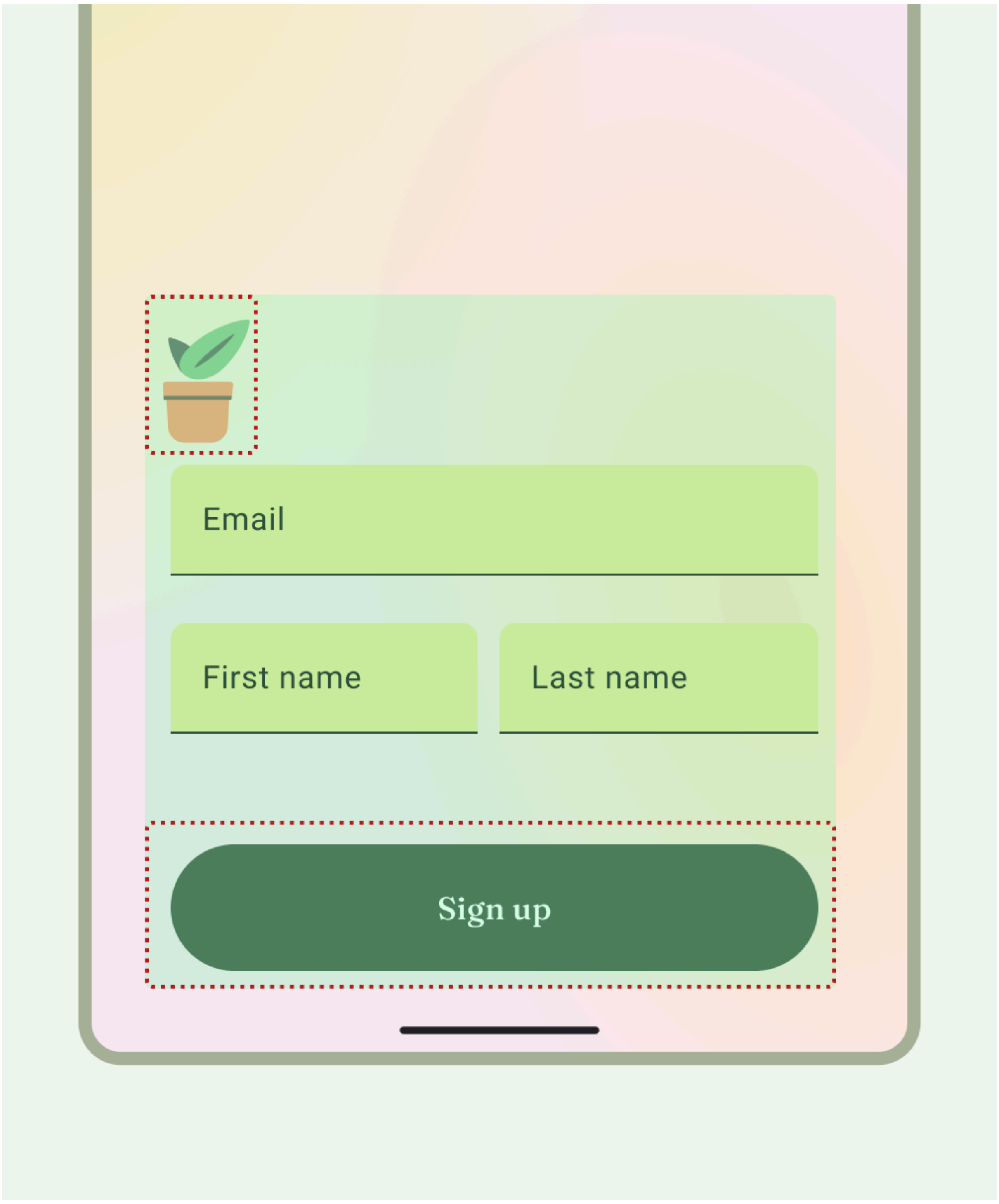


Figure 12: *Layout example showing containment boundaries and position of elements*

Gravity is a standard for placing an object within a potentially larger container for specific use cases. The following figure shows examples of positioning objects start and center ( ), top and center horizontal ( ), bottom left ( ), and end and right ( ).

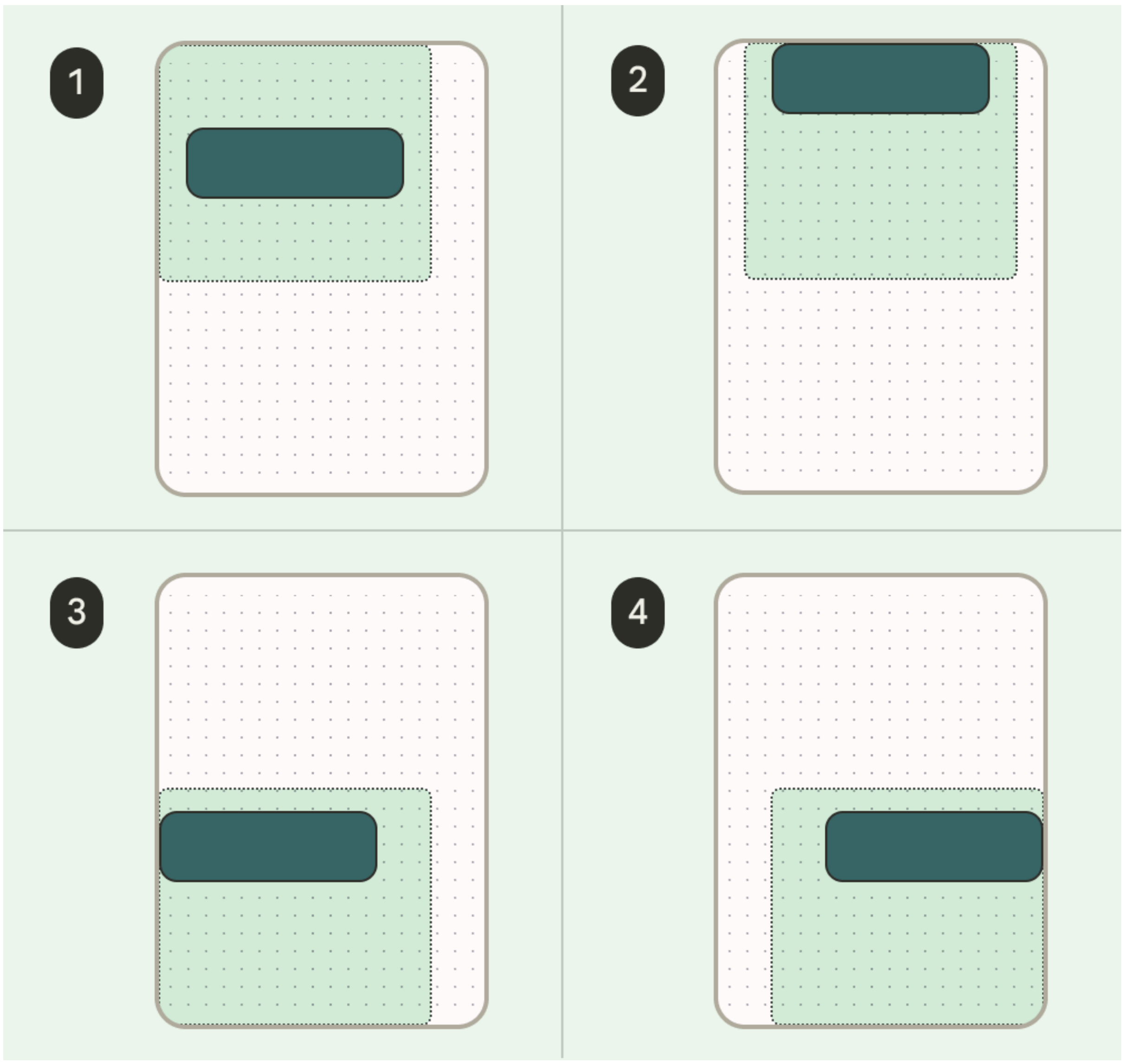


Figure 13: *Positioning gravity of child content and parent views*

Scaling

Scaling is crucial to accommodate dynamic content, device orientation, and screen sizes. Elements can remain fixed or be scaled.

Noting how images are displayed within their containers with scaling and position is important to ensure it appears how you want the image to look despite the device context, otherwise the image's primary focus could appear cut off, images could be too small or large for the layout, or other undesirable effects.

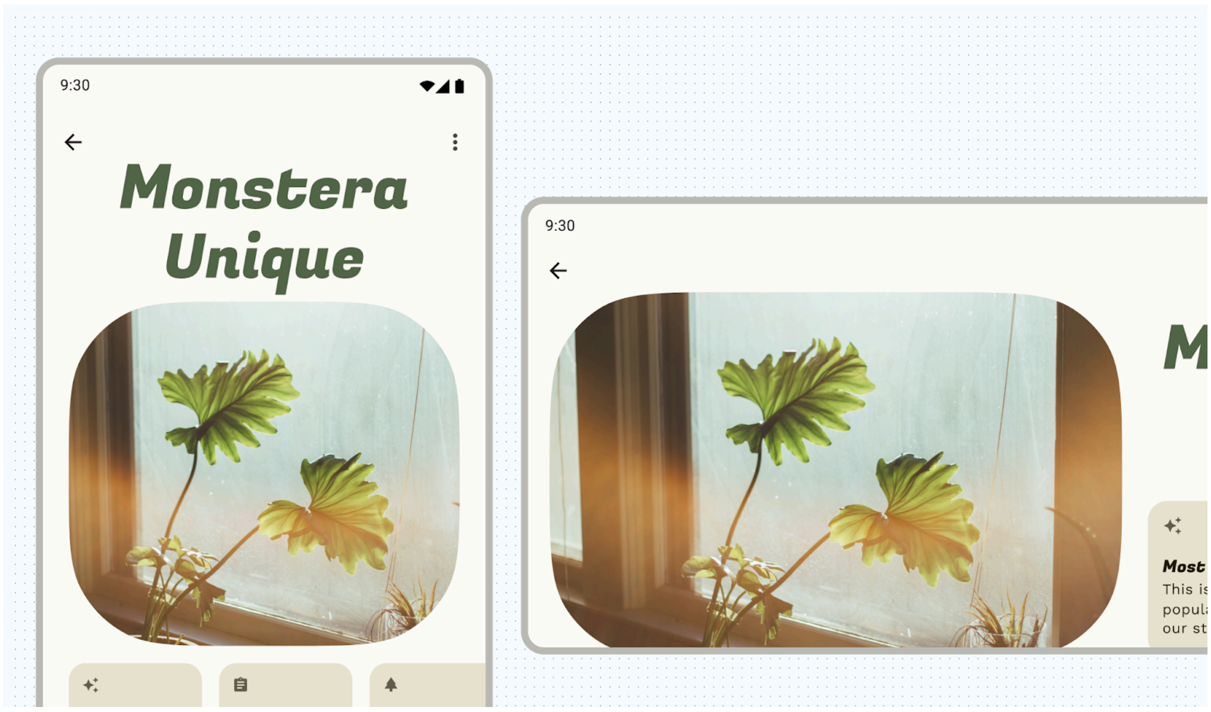


Figure 14: *Center-cropped image, which ensures the plant is centered within the container regardless of device size*

Content that is not notated can appear differently than you expect or want.

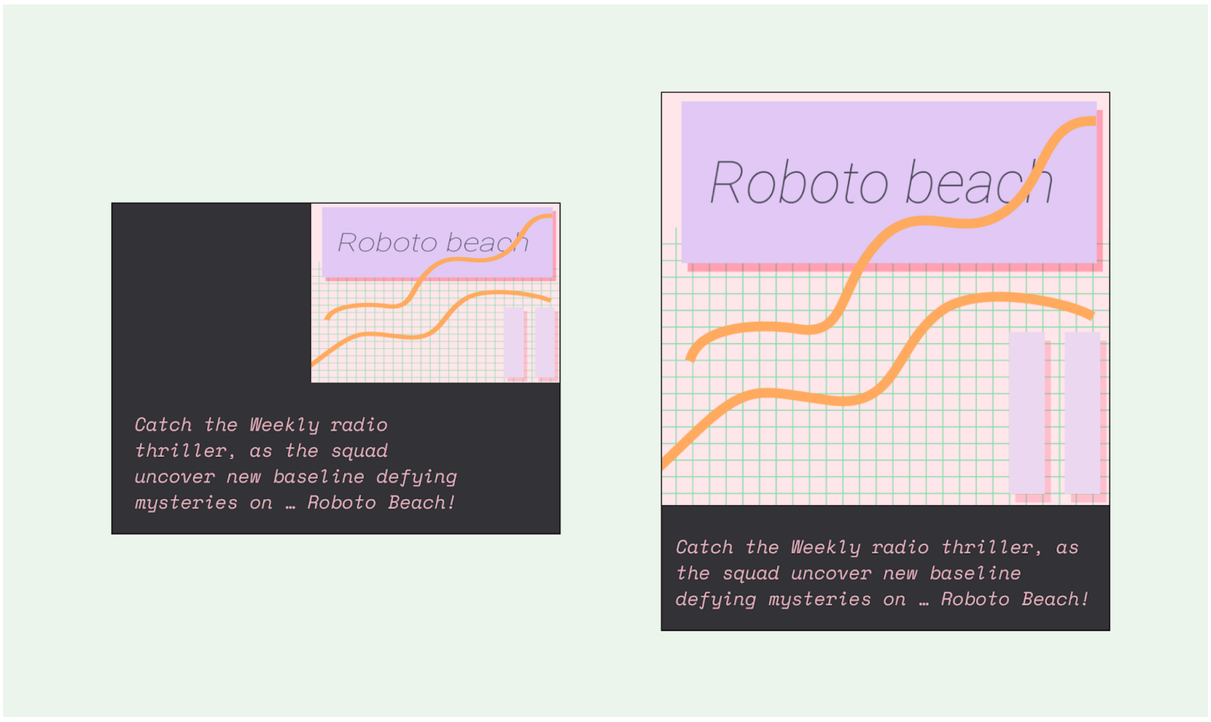

Roboto beach
Catch the Weekly radio thriller, as the squad uncover new baseline defying mysteries on … Roboto Beach!
Roboto beach
Catch the Weekly radio thriller, as the squad uncover new baseline defying mysteries on … Roboto Beach!

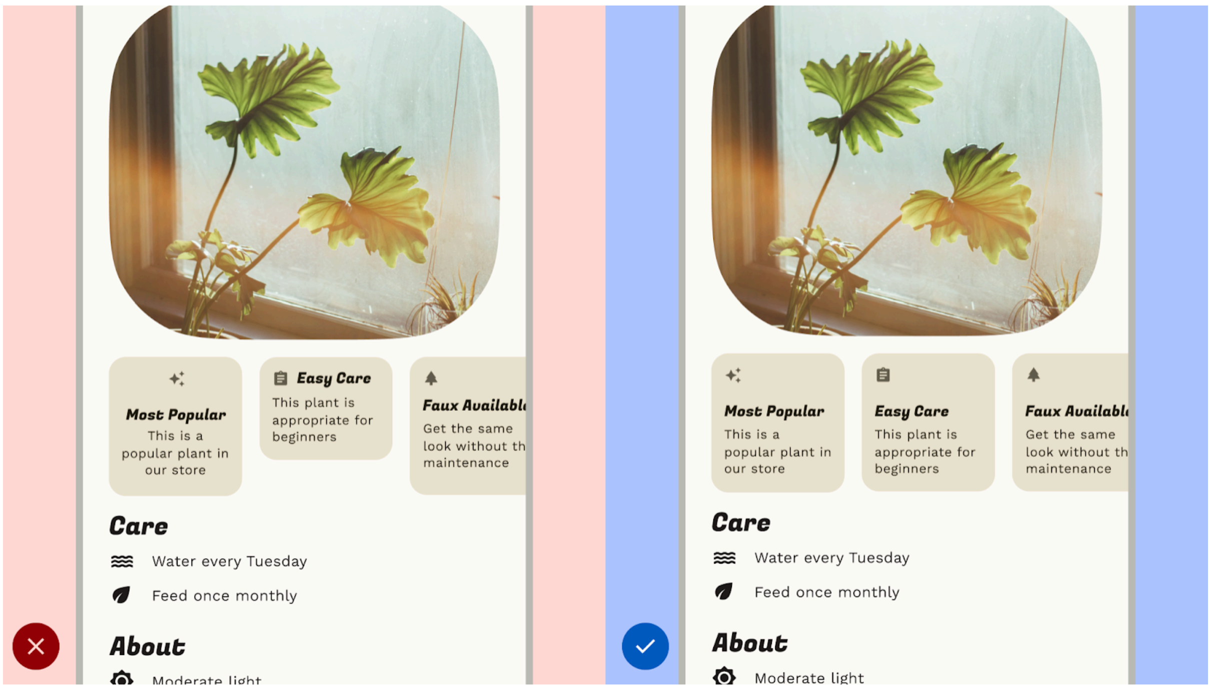
Figure 15: *Content that is not notated can appear in unexpected ways*

Pinned content

Many elements have built in interactions, scrolling, and positioning with slots or scaffolds. Some elements can be modified to stay fixed instead of reacting to scrolling, for example floating action buttons (FABs) that house critical actions.

Alignment

Use `AlignmentLine` to create custom alignment lines, which parent layouts can use to align and position their children.



Most Popular
This is a popular plant in our store
Easy Care
This plant is appropriate for beginners
Faux Available
Get the same look without the maintenance
Care
Water every Tuesday
Feed once monthly
About
Moderate light

***Figure 16:** Don't disrupt readability*

Don't: disrupt readability by inconsistently spacing like elements, which can make designs appear haphazard.

Do: Establish consistent spacing between like elements.

Component layout

Material 3 components provide their own configurations and states for interaction and content.

Compose provides convenient layouts for combining Material Components into common screen patterns. Composables such as Scaffold provide slots for various components and other screen elements. [Read more about Material Components and Layout](#).

Layouts and navigation patterns

If your app contains multiple destinations for users to traverse, we recommend employing layout and navigation pairings that are commonly used by other apps. Because many users already possess the mental models for these pairings, your app will be more intuitive for them.

Layout and navigation pairings

The navigation bar and modal navigation drawer are used as primary navigation patterns for parent layout views and primary navigation destinations.

The navigation bar can hold three to five navigation destinations across the same hierarchy level. This component translates to the navigation rail for large screens.

Although the navigation drawer can hold more than five navigation destinations, the pattern is not as ideal as the navigation bar due to the need to reach to the top bar on compact sizes.

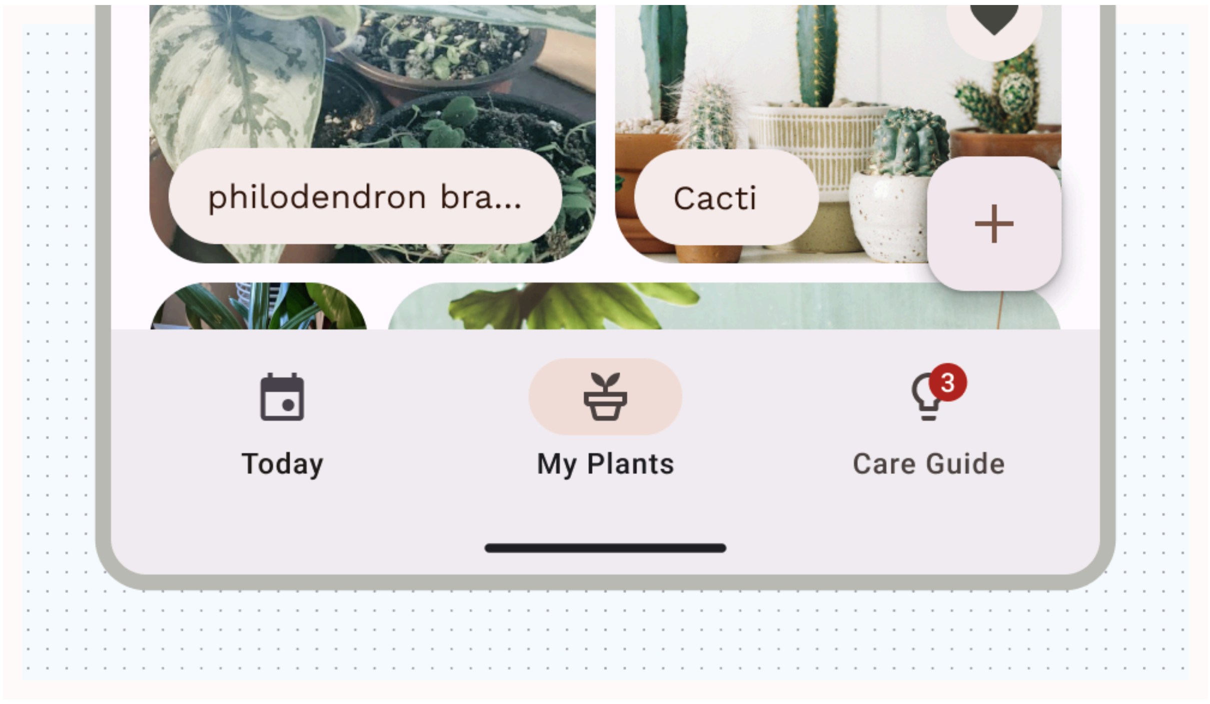


Figure 17: *Primary navigation destinations within a navigation bar*

Material 3 Tabs and the bottom app bar are secondary navigation patterns that you can can use to supplement primary navigation or appear on children views.

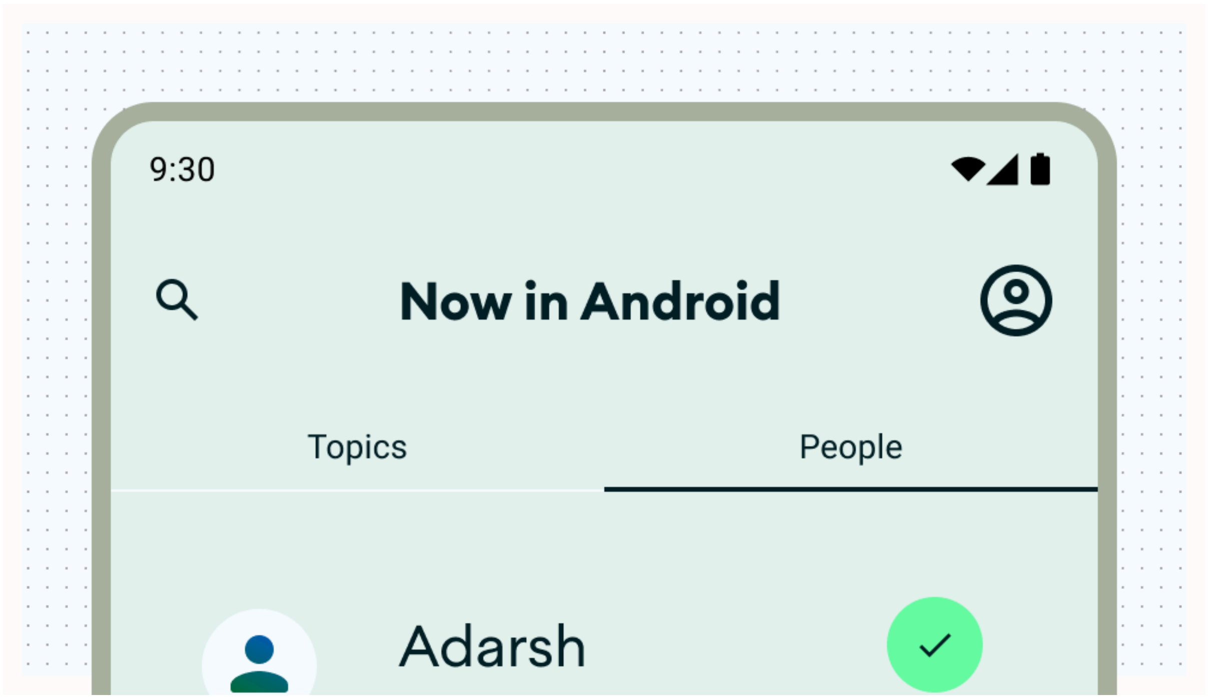


Figure 18: *Tabs act as a secondary navigation layer to group sibling content (secondary)*

Layout actions

Provide controls to enable users to accomplish actions. Common patterns include top bar actions, floating action button (FAB), and menus.

For actions of the highest importance, a FAB provides a large and prominent button for the user. Provide only one action at a time at this level. A FAB can appear in multiple sizes and an expanded form, which includes a label. Use `Scaffold` to pin a FAB, making sure it's always visible even on scroll by.

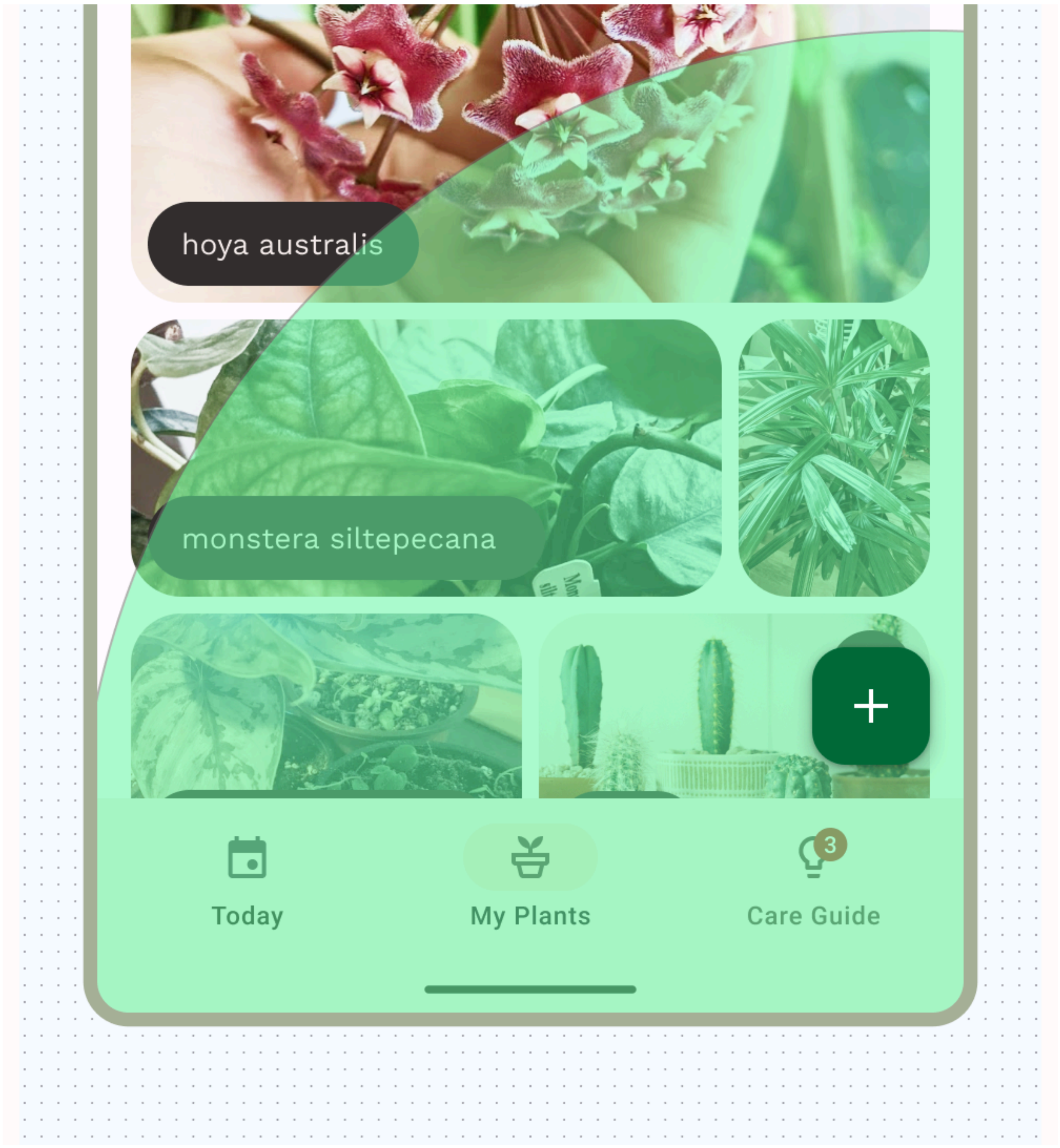


***Figure 19:** A floating action button (FAB) that allows the user to quickly add plants to the plant gallery*

You can place secondary actions within the top bar or, if it's grouped near related content, within the page.

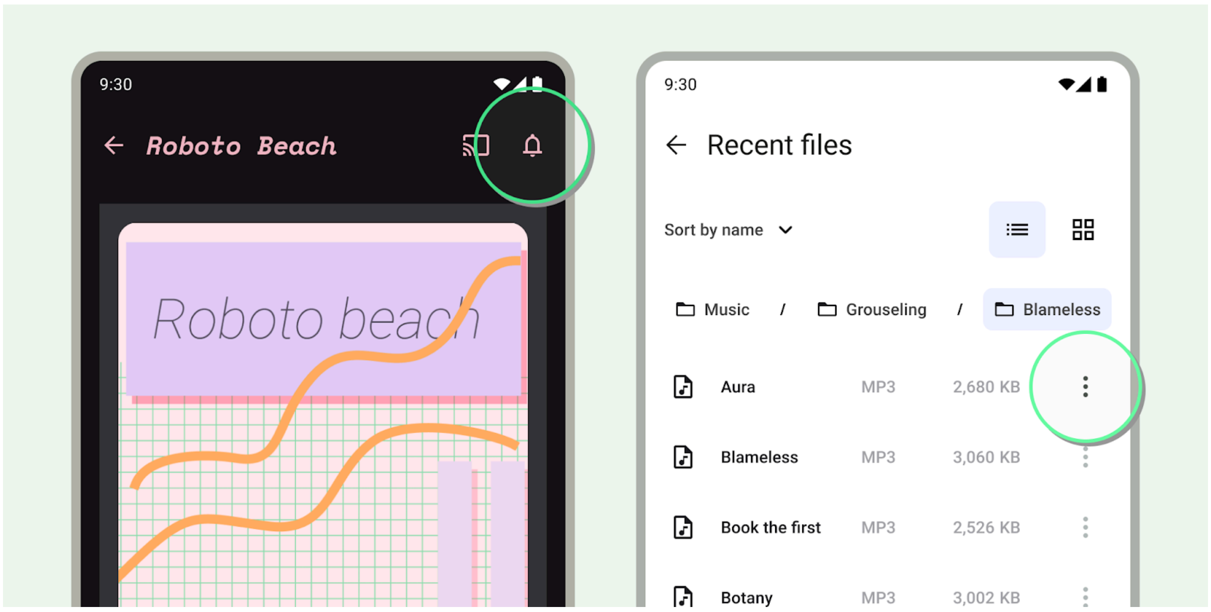


Figure 20: *Alert action in the top bar on show detail (left) and overflow icon in line of list item (right)*

For any additional actions that aren't promptly or frequently needed, add those actions in an overflow menu.

Canonical layouts

Utilize canonical layouts as a starting point, ready-to-use compositions that help layouts adapt for common use cases and screen sizes. These layouts are aesthetic and functional, and derived from [Material 3 guidance](https://m3.material.io/).

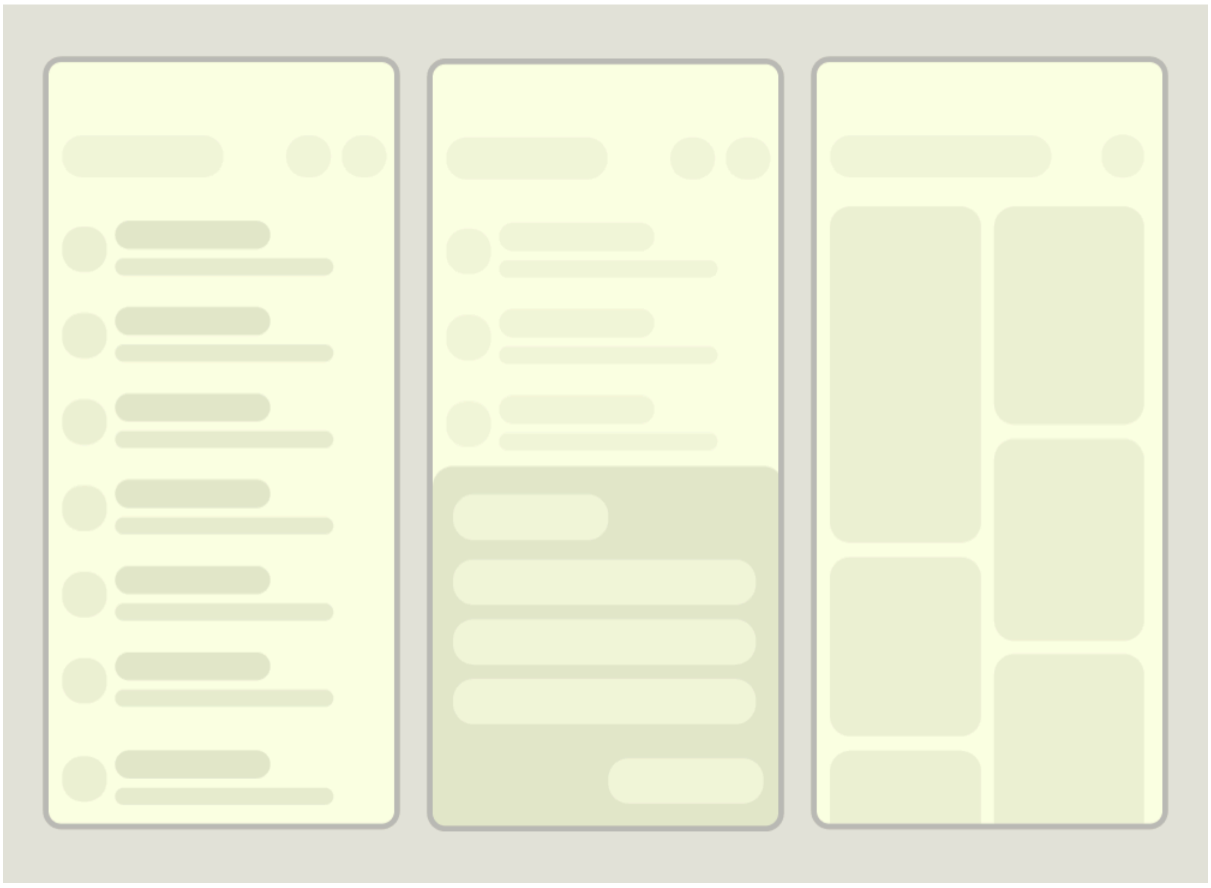


Figure 21: *Canonical layouts*

The Android framework includes specialized components that make implementation of the layouts straightforward and reliable using either Jetpack Compose or Views APIs.

List-detail layout

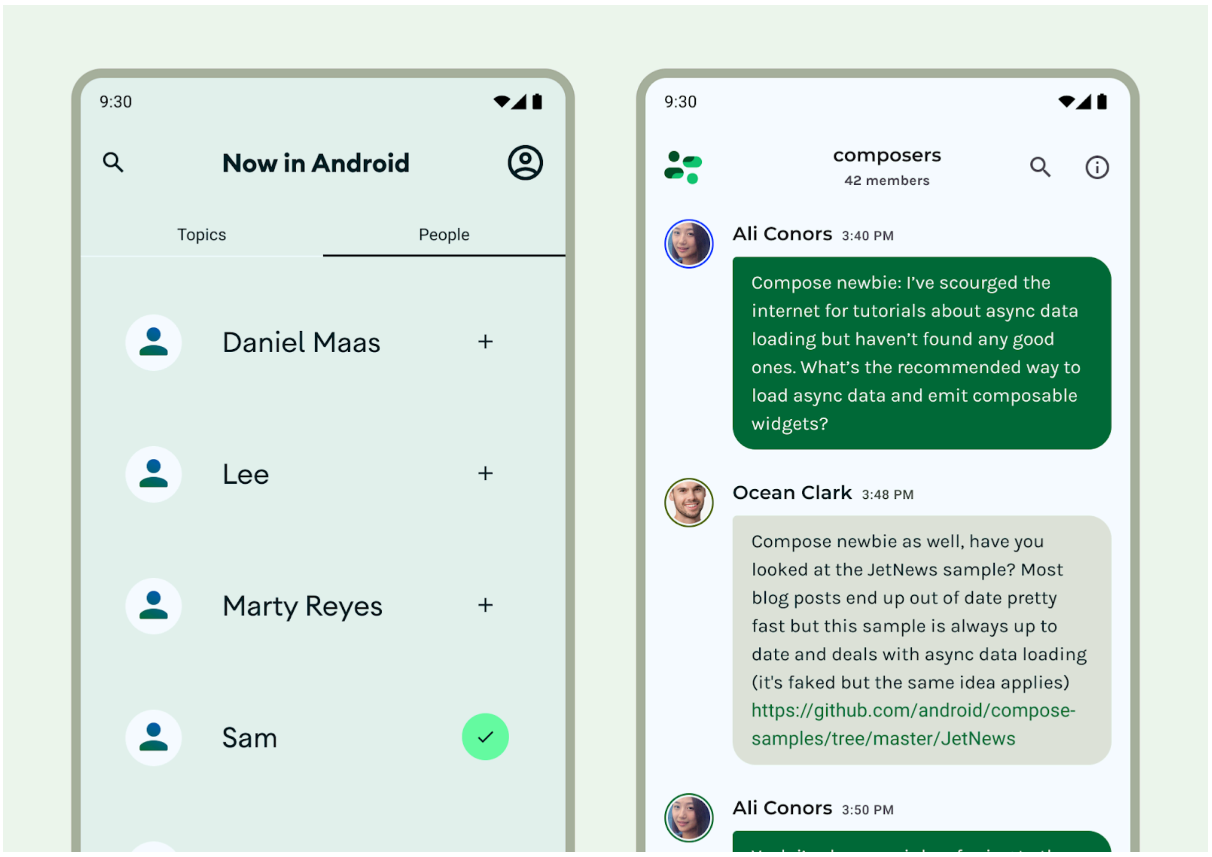
A list-detail layout enables users to explore lists of items that have descriptive, explanatory, or other supplementary information—the item detail. For compact screen sizes, only the list or detail view are visible. Displaying a collection of content in a row-based layout, lists make up the most common form of layouts for apps. List-detail is ideal for messaging apps, contact managers, file browsers, or any app where the content can be organized as a list of items that reveal additional information.

Content can be static or dynamic.

- Dynamic content is content that your app serves on-the-fly, and is ideal for showing user-generated content or reflect the user's preference or actions. For example, imagine a photo app with a scrollable list of user-generated

photos, which is unique for each user and changes as the user uploads more images. These images are dynamic content.
- Static content represents hard-coded content, which is modifiable only by making changes directly to your app's code. Examples of static content are images and text that every user might see.

The Now in Android Figma file provides multiple layout examples. The following example shows a one-dimensional collection of content.



Ali Conors 3:40 PM
Compose newbie: I've scourged the internet for tutorials about async data loading but haven't found any good ones. What's the recommended way to load async data and emit composable widgets?

Ocean Clark 3:48 PM
Compose newbie as well, have you looked at the JetNews sample? Most blog posts end up out of date pretty fast but this sample is always up to date and deals with async data loading (it's faked but the same idea applies) https://github.com/android/compose-samples/tree/master/JetNews

Ali Conors 3:50 PM

Figure 22: *One dimensional collection of content*

Explore Material 3 Lists for more design guidance on list components and specs.

Feed layout

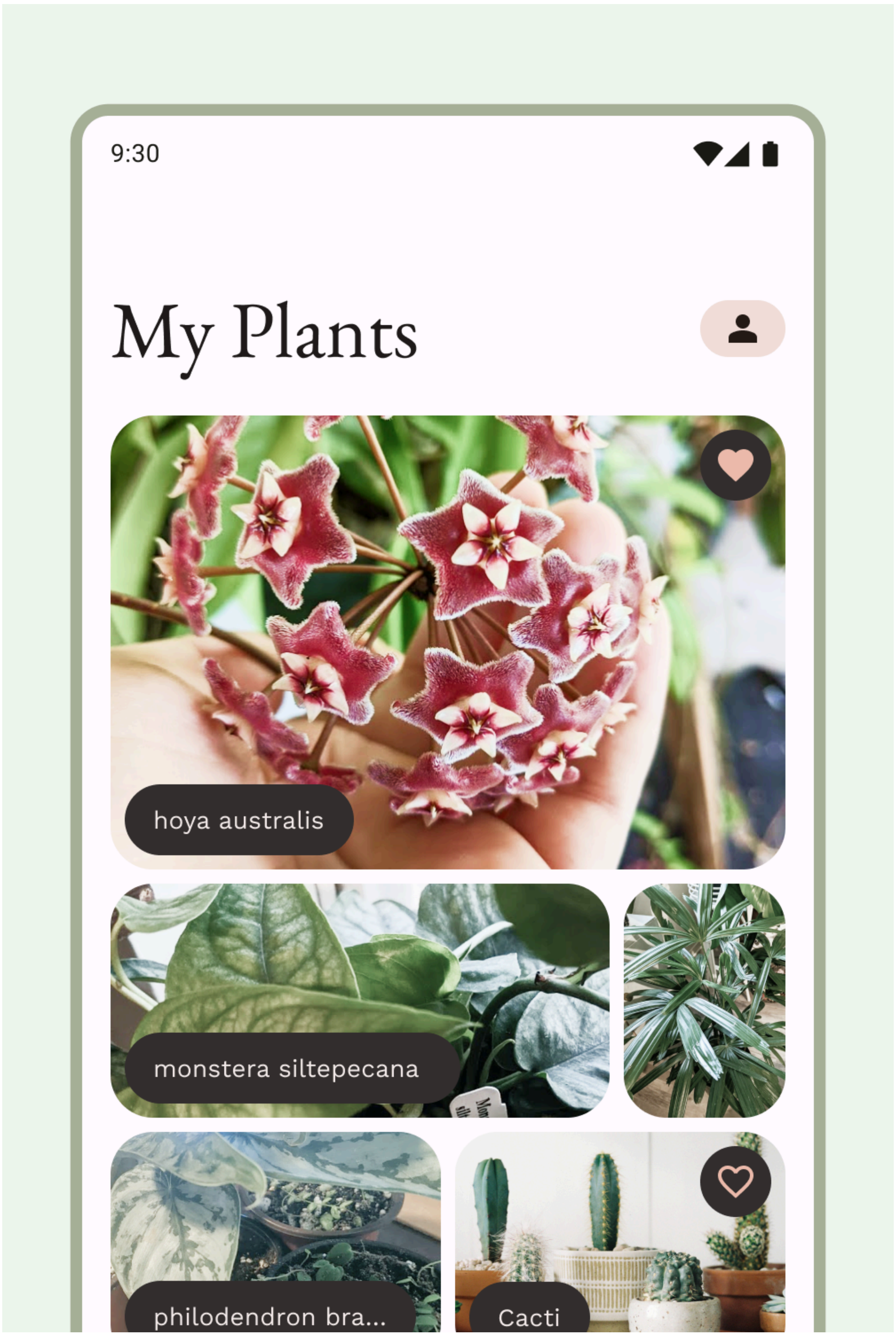


Figure 23: *A photo gallery in a grid layout is a common feed format*

A feed layout arranges equivalent content elements in a configurable grid for quick, convenient viewing of a large amount of content. (See Material 3 guidelines for using cards in a collection.) Feeds can be list- or grid- based configuration on compact displays, typically in cards or tiles. Content can be dynamic, meaning it is “fed in” from a dynamic external source such as an API.

A grid layout is composed of both rows and columns made up by implied or explicit containment principles. (See containment on this page for more information.) A grid layout can be more rigidly applied or staggered to vary the rows and columns. Both should have consistent application of spacing and logic to avoid confusing users. Explore Material 3 guidelines about designing feeds.

You can implement a feed layout in Compose with Lazy lists or lazy grids, or in Views with `RecyclerView` or `CardView`.

Supporting pane layout

A mobile view may require supporting content or controls. Typically in the form of sheets or dialogs, they can help the primary view stay focused and uncluttered. Check out M3 guidance for using the supporting pane canonical layout.

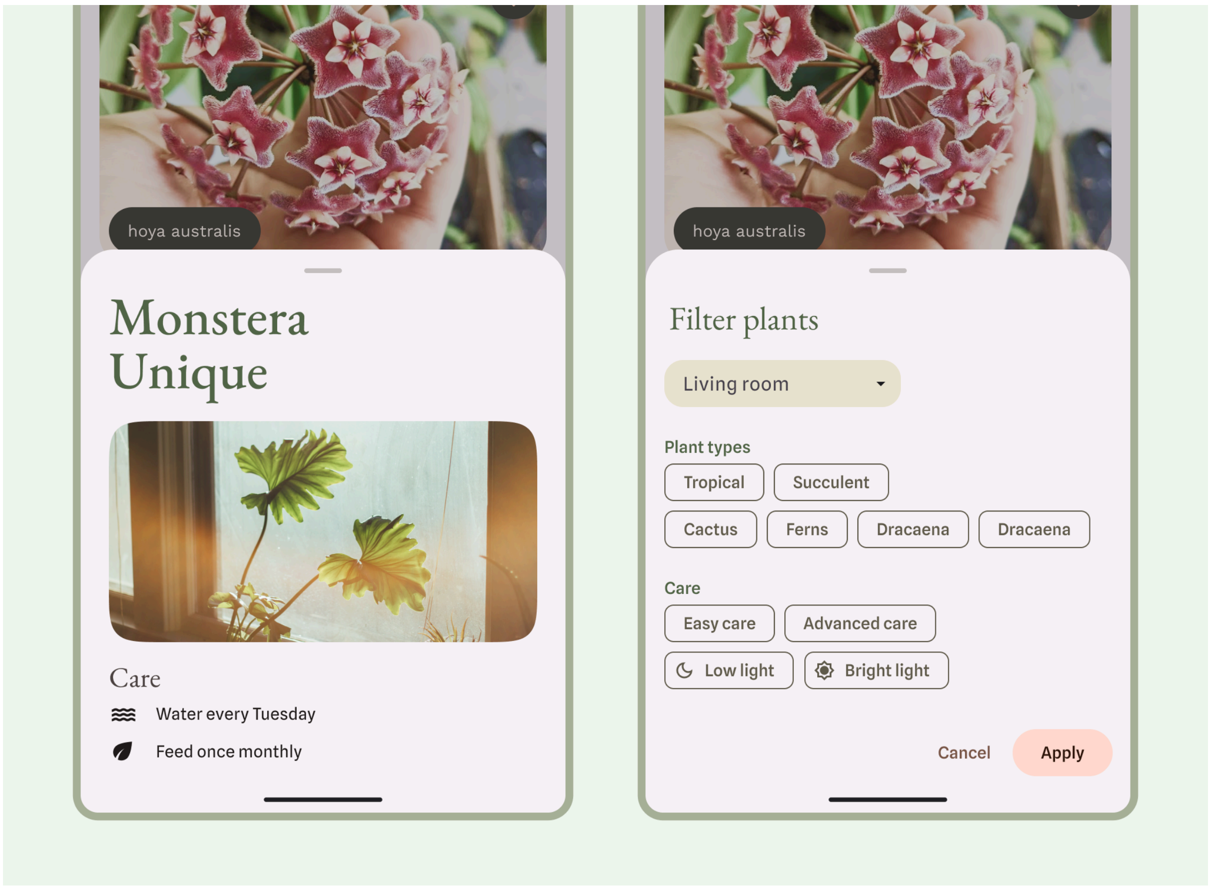


Figure 24: *Bottom sheets can act as supporting content to the primary view*

Learn about M3 guidance for bottom sheets.

Relative layouts

Inputs, content, or other actions may appear relative to each other or constrained to a parent container. Layouts can be more custom, but make sure to follow consistent grouping, columns, and spacing.

Layouts can also use a combination of layout types. For example, you might pair a carousel or horizontal scroll with vertical cards. Or, you could present a custom chart with vertical list data.

You can present content in scrolling rows or columns with lazy rows and lazy columns.

Learn more about Compose layout basics and what makes up a composable.

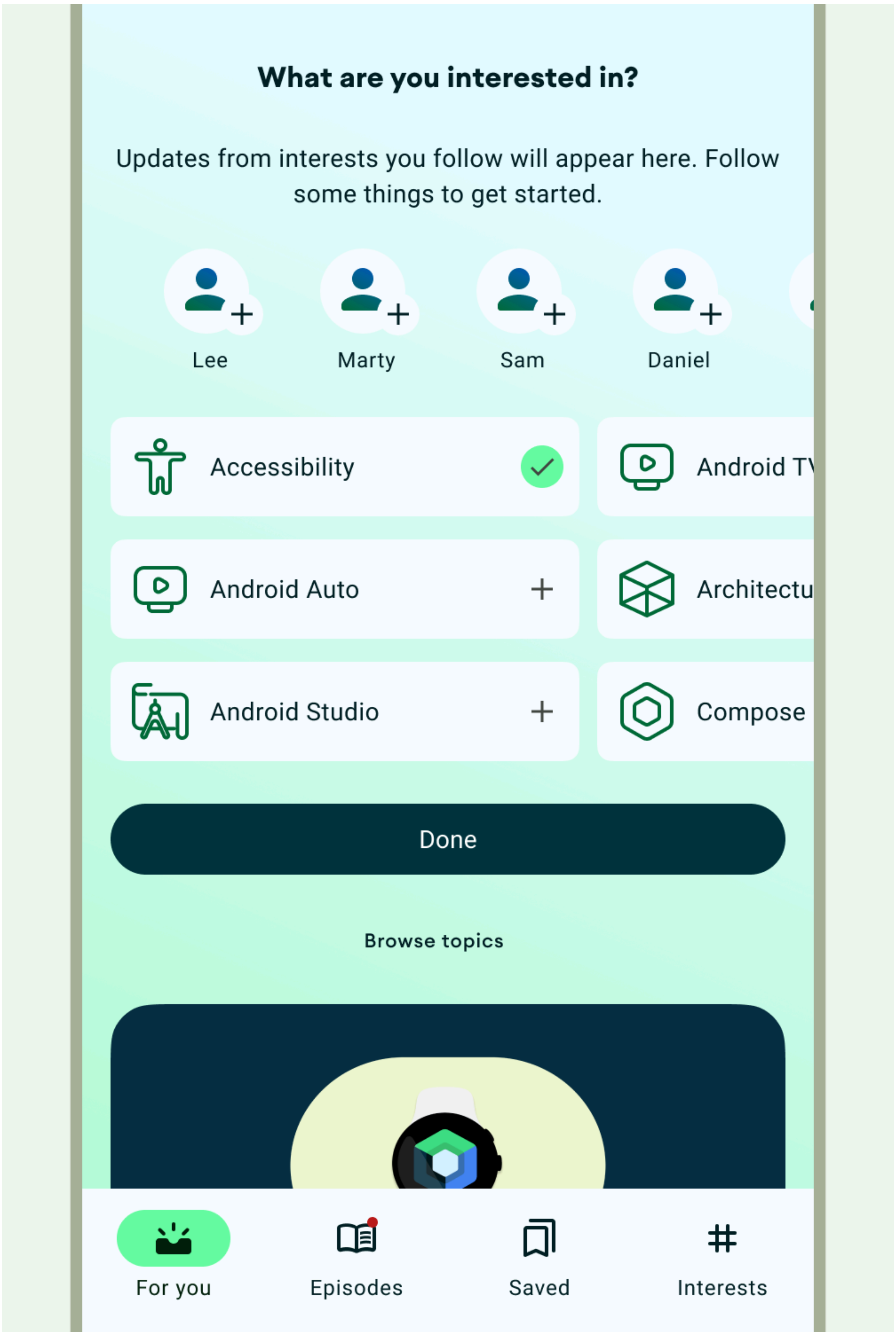


Figure 25: *Layouts can have a combination of groupings, lists, and grids*

Authentication is a common relative layout, as shown in the following figure.

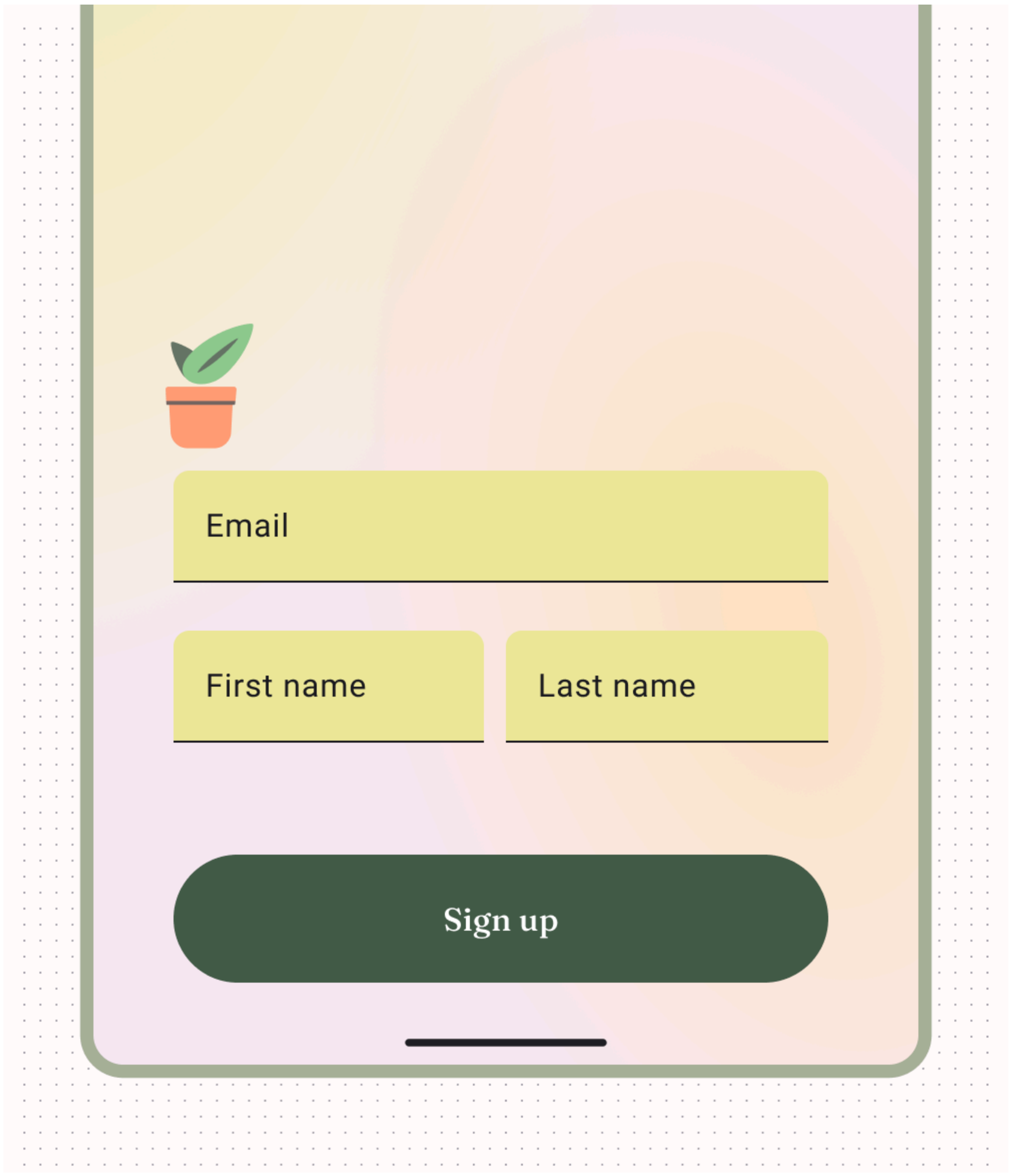


Figure 26: *Authentication as a common layout*

Full-screen layout is another common layout, as used in immersive mode.

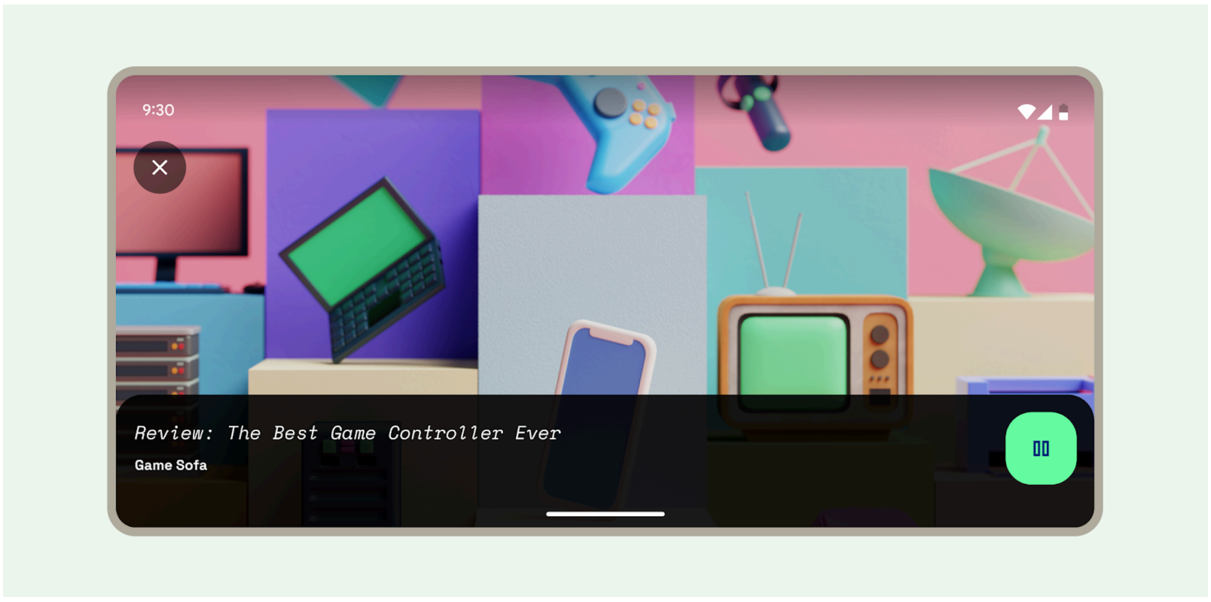
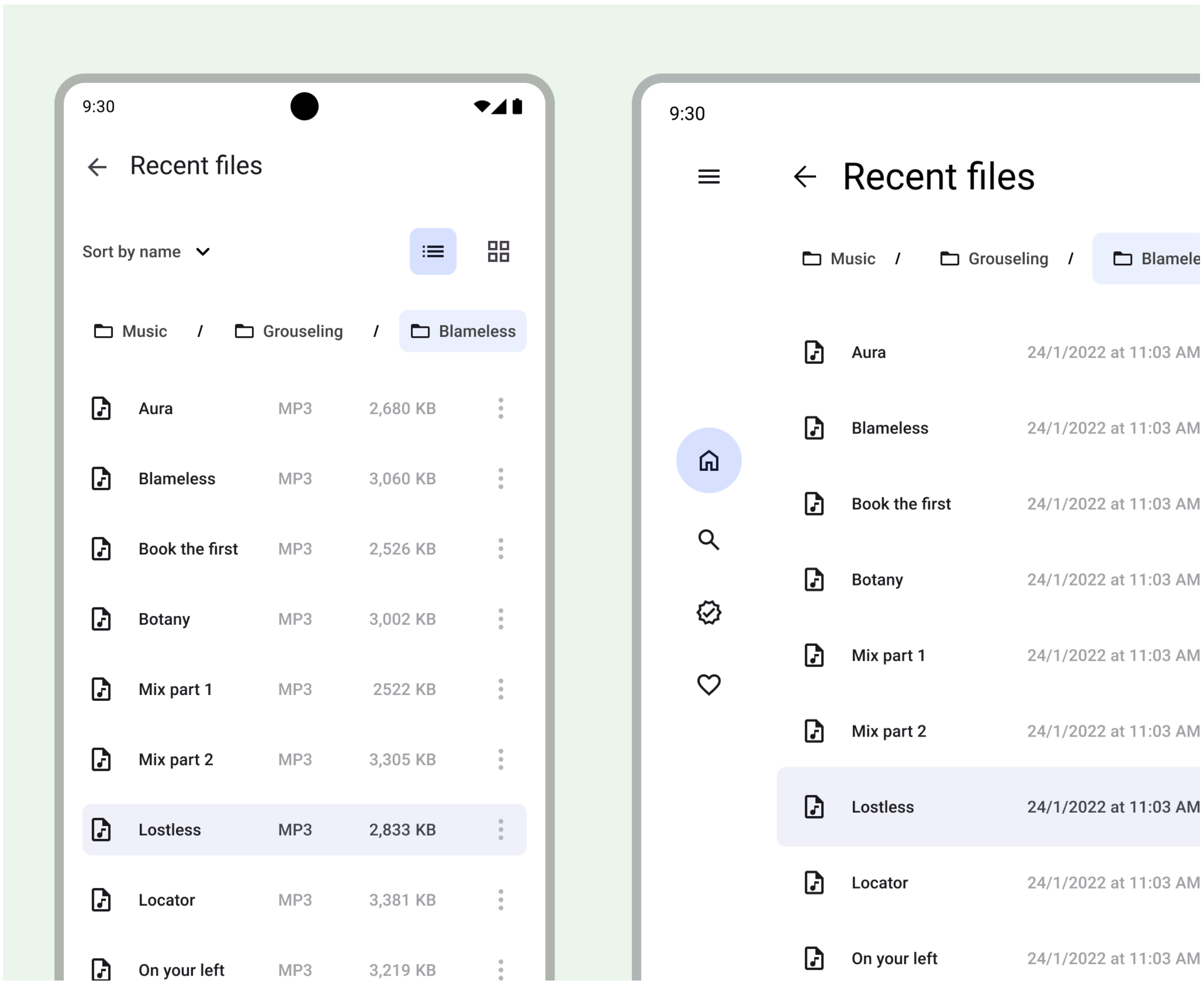


Figure 27: *Full screen layout, as used in immersive mode*

If you're working with Views instead of Compose, you can use `ConstraintLayout` to lay out views according to relationships between sibling views and the parent layout, allowing for large and complex layouts. `ConstraintLayout` lets you build entirely by dragging and dropping instead of editing the XML using the layout editor. Learn more about building a UI with Layout Editor.

Adapt layouts

Adaptive design is the practice of designing layouts that adapt to specific breakpoints and devices. Usually we consider the width of the device to determine where the layout should change, or adapt. Both Web and Android utilize responsive design concepts, like flexible grids and images, to create layouts that better respond to their context.



Aura	MP3	2,680 KB
Blameless	MP3	3,060 KB
Book the first	MP3	2,526 KB
Botany	MP3	3,002 KB
Mix part 1	MP3	2522 KB
Mix part 2	MP3	3,305 KB
Lostless	MP3	2,833 KB
Locator	MP3	3,381 KB
On your left	MP3	3,219 KB

Aura	24/1/2022 at 11:03 AM
Blameless	24/1/2022 at 11:03 AM
Book the first	24/1/2022 at 11:03 AM
Botany	24/1/2022 at 11:03 AM
Mix part 1	24/1/2022 at 11:03 AM
Mix part 2	24/1/2022 at 11:03 AM
Lostless	24/1/2022 at 11:03 AM
Locator	24/1/2022 at 11:03 AM
On your left	24/1/2022 at 11:03 AM

For design guidelines about adapting layouts to expanded screen sizes, read the Build adaptive layouts developer guide in Compose and the M3 Applying Layout page. You can also check out the Android large screen canonical gallery for inspiration and implementation of large screen layouts.

Although not every app needs to be available on every screen size, it does allow your users more freedom regarding ergonomics, usability, and app quality.

- You can design key screens (communicate the essential concepts or your app) with class sizes as breakpoints to act as guidelines.
- Or design content to act responsively by notating how content should be constrained, expand, or reflow.

For more on layouts, check out the Material Design 3 (M3) Understanding layout page.