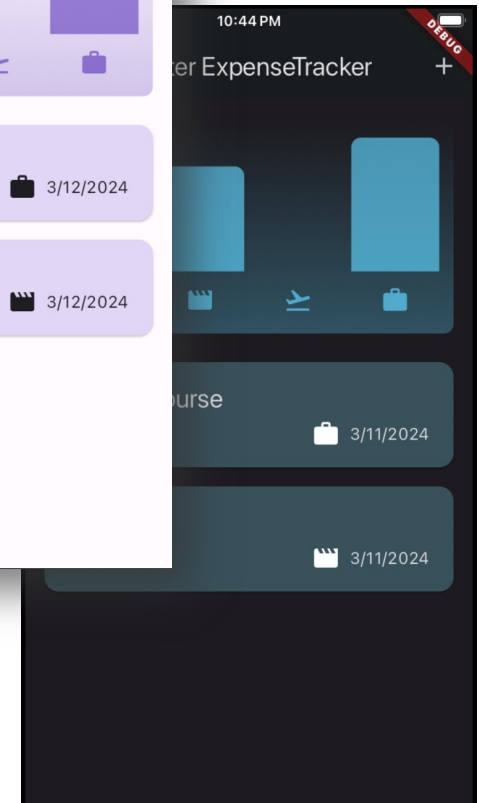
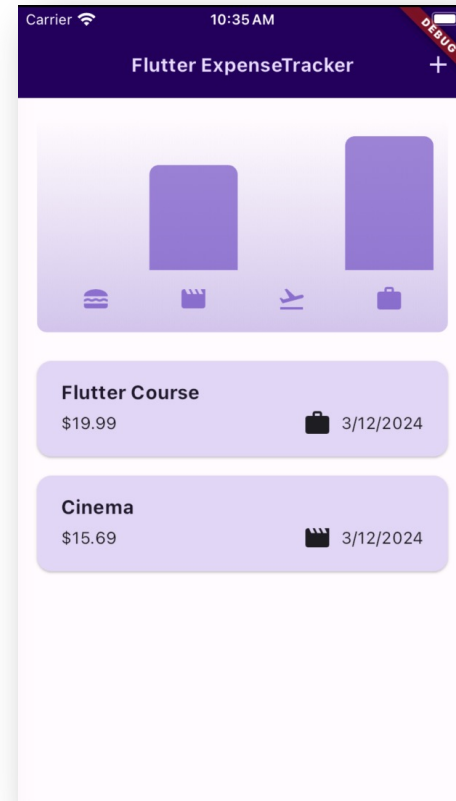


Rendering & Responsive UI

Shan-Hung Wu
CS, NTHU

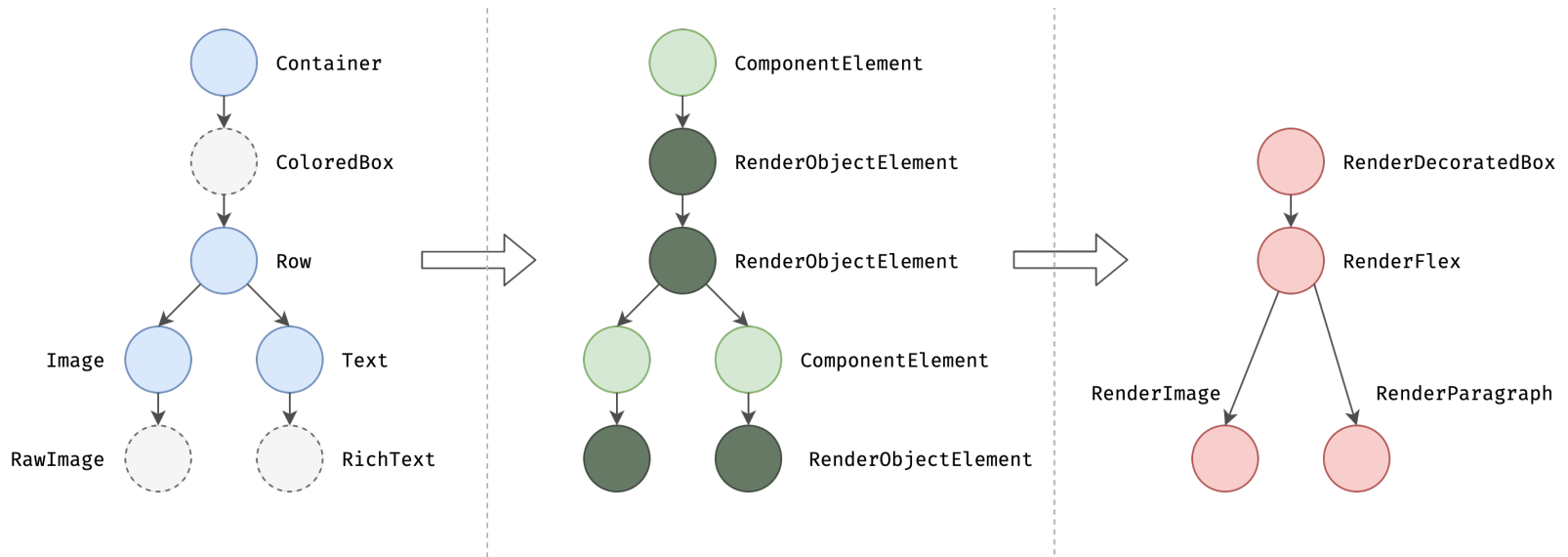
Today's Topics

- Build & rendering
- Element tree
- Render tree
- Responsive layout
- Custom painting & parallax scrolling



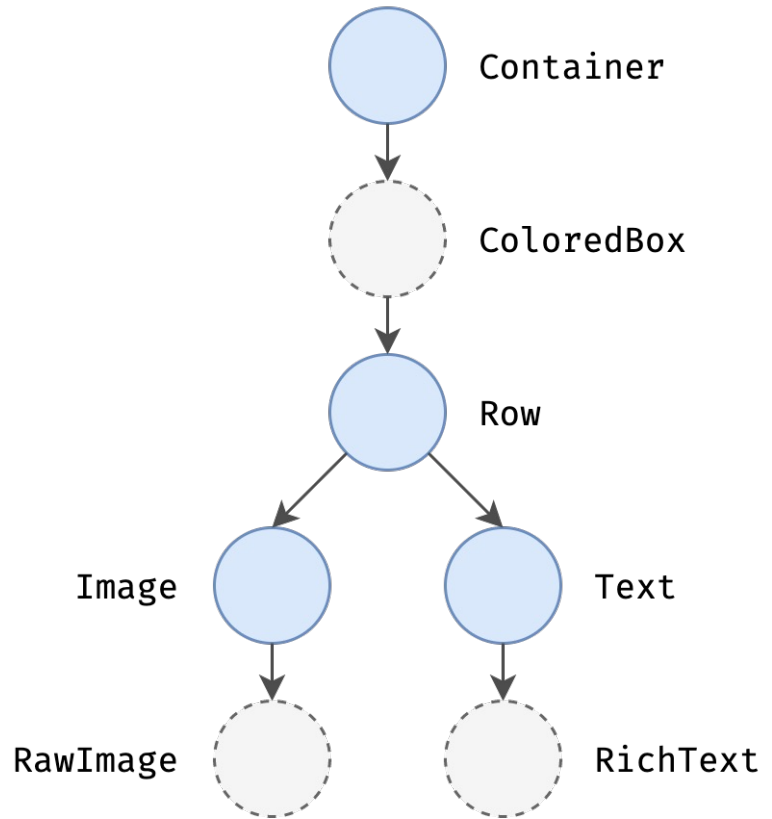
Build & Rendering Process

- **Widget tree**: UI declaration
- **Element tree**: states & build context
- **Render tree**: layout & rendering



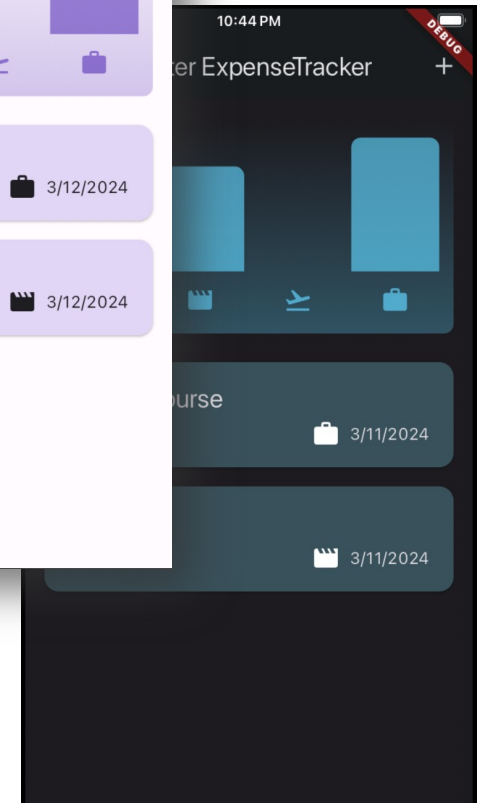
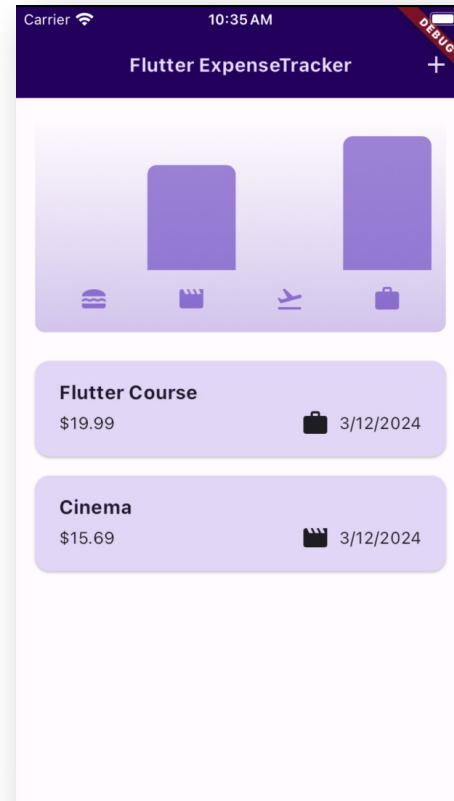
Widget Tree

- For each stateful widget, its child widgets tree are reconstructed whenever state changes
- Keep your stateful widget as small as needed
 - Extract stateful subtree to new stateful widget
 - Favor composition over inheritance



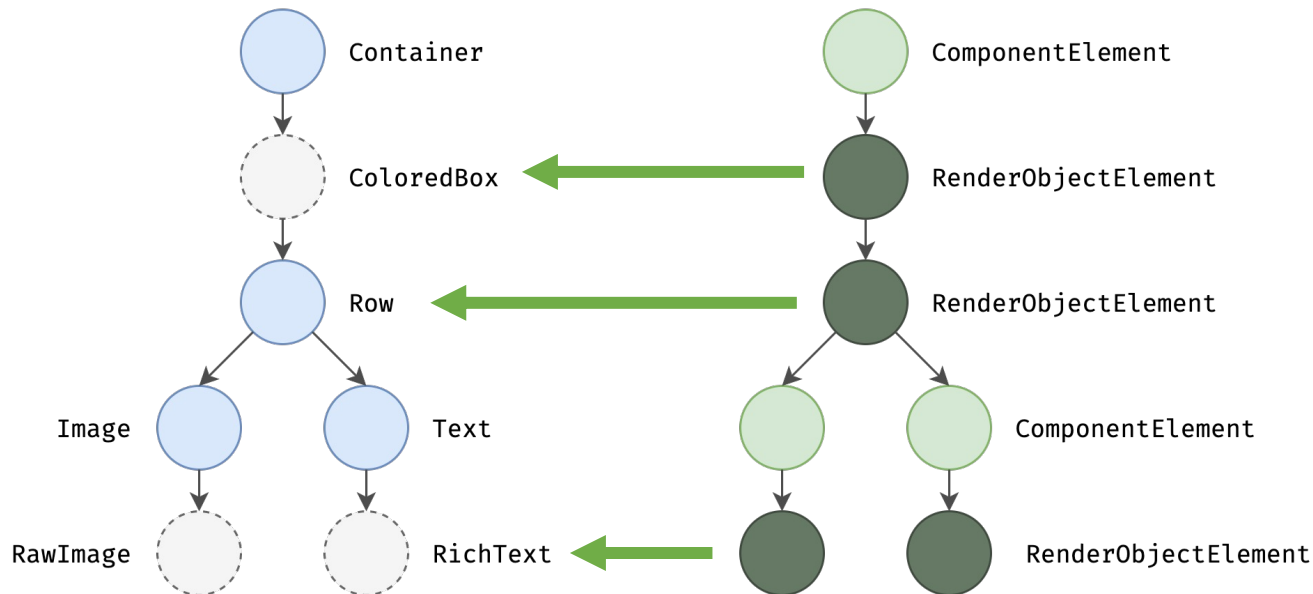
Today's Topics

- Build & rendering
- Element tree
- Render tree
- Responsive layout
- Custom painting & parallax scrolling



Element Tree & Node

- Every widget has `createElement()` method
- Each element node “points to” corresponding widget
- Provides ***build context*** and ***state*** for the widget



Widget are Short-lived

- Element rebuilds widget if state (local data) changes
- Why get global data this way?

```
Theme.of(context)
```

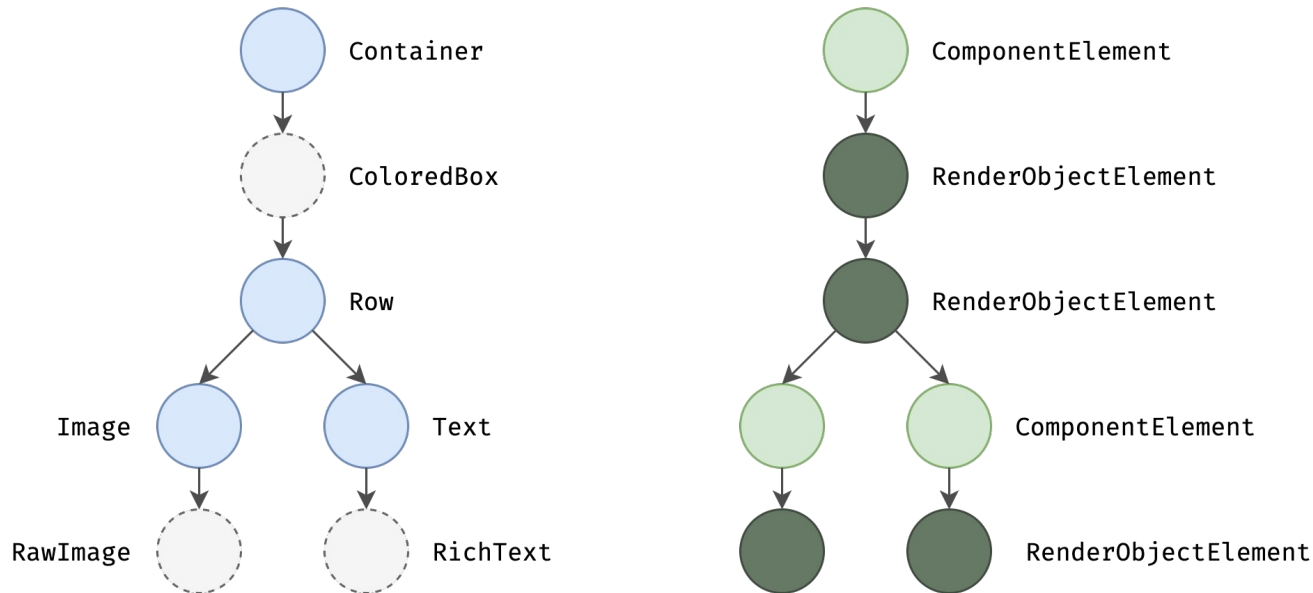
```
MediaQuery.of(context)
```

```
Navigator.of(context)
```

1. Start from element associated with context
2. Moving up along element tree to find nearest InheritedWidget (IW) of **matching type**
3. Register element as listener of IW and forward global data from IW to widget
4. **Rebuild widget** whenever global data change

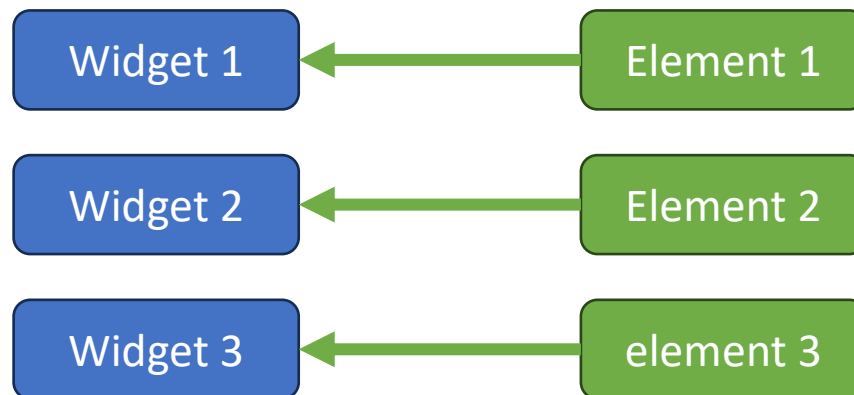
Element Nodes are Long-lived

- `initState()` and `dispose()` **not** called on every widget reconstruction
- How to map widget node to element node?



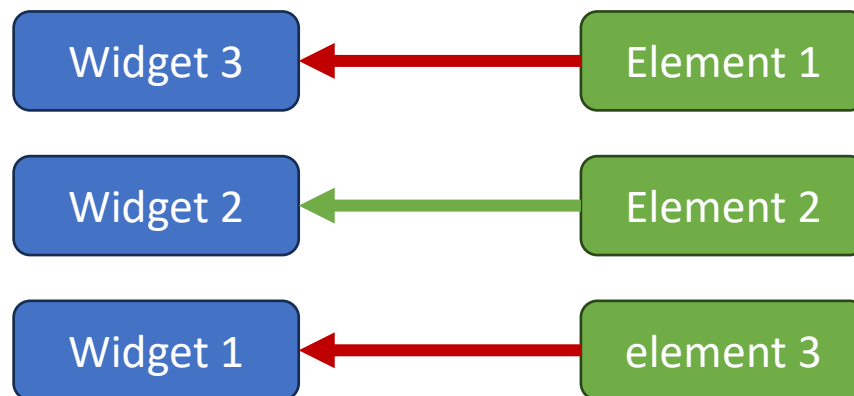
Node Mapping

- By default, element node checks the *class* of widget node to detect changes



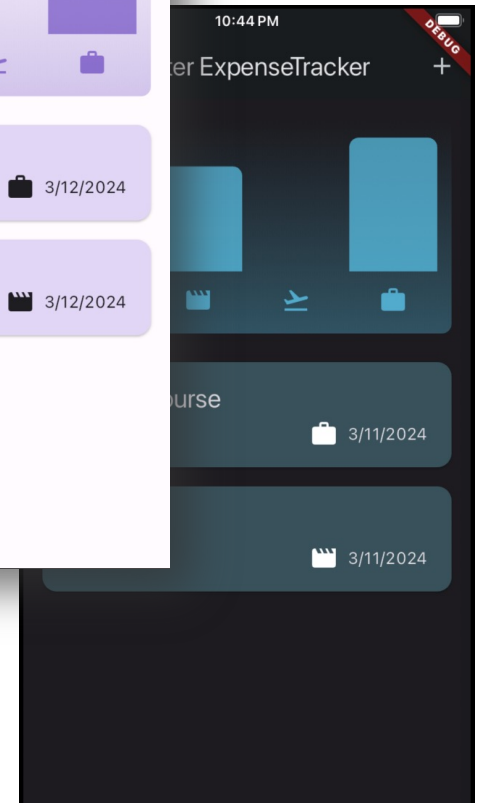
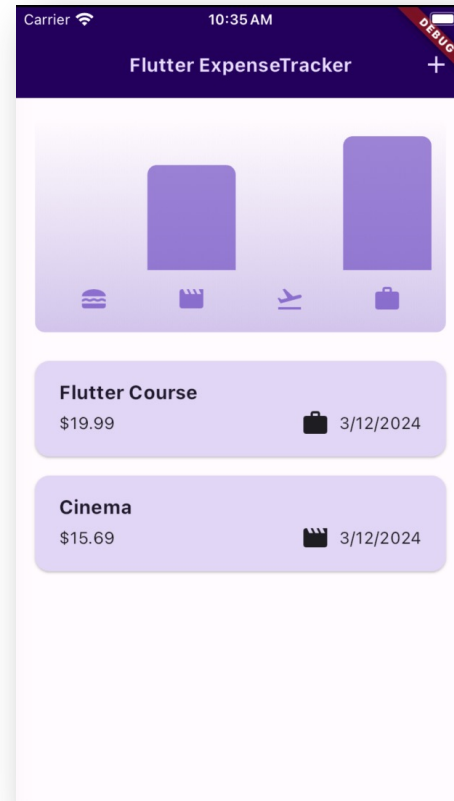
Node Mapping

- By default, element node checks the *class* of widget node to detect changes
- Not ideal for changing list items of the same type
- If `Key` available, element node use it to detect changes

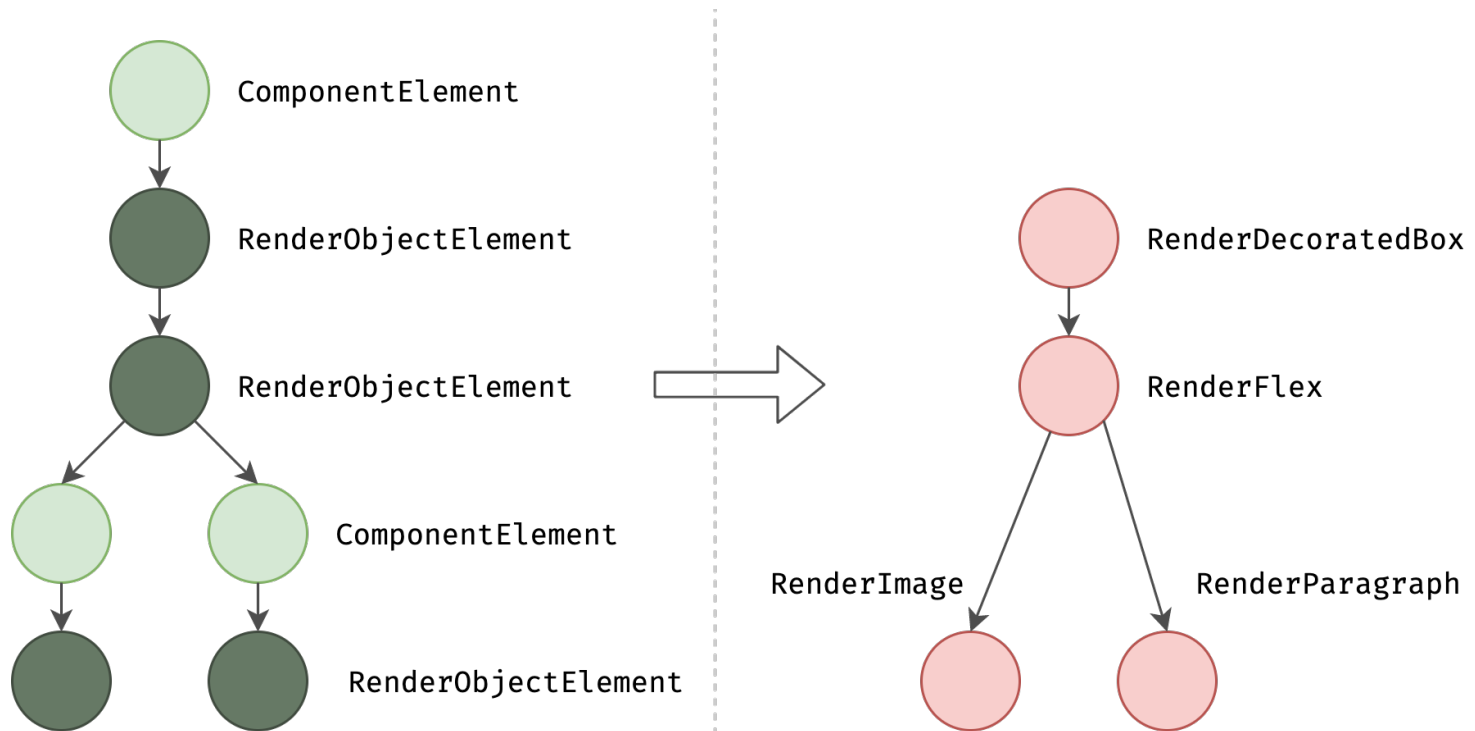


Today's Topics

- Build & rendering
- Element tree
- **Render tree**
- Responsive layout
- Custom painting & parallax scrolling

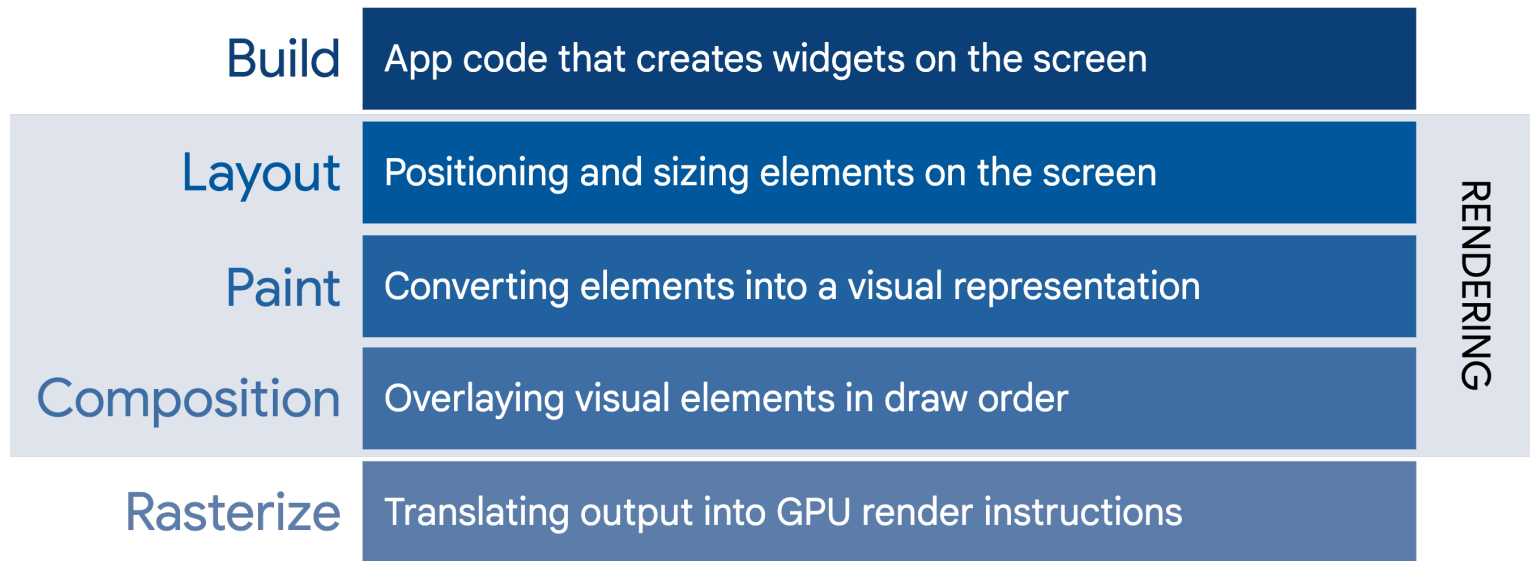


Render Tree



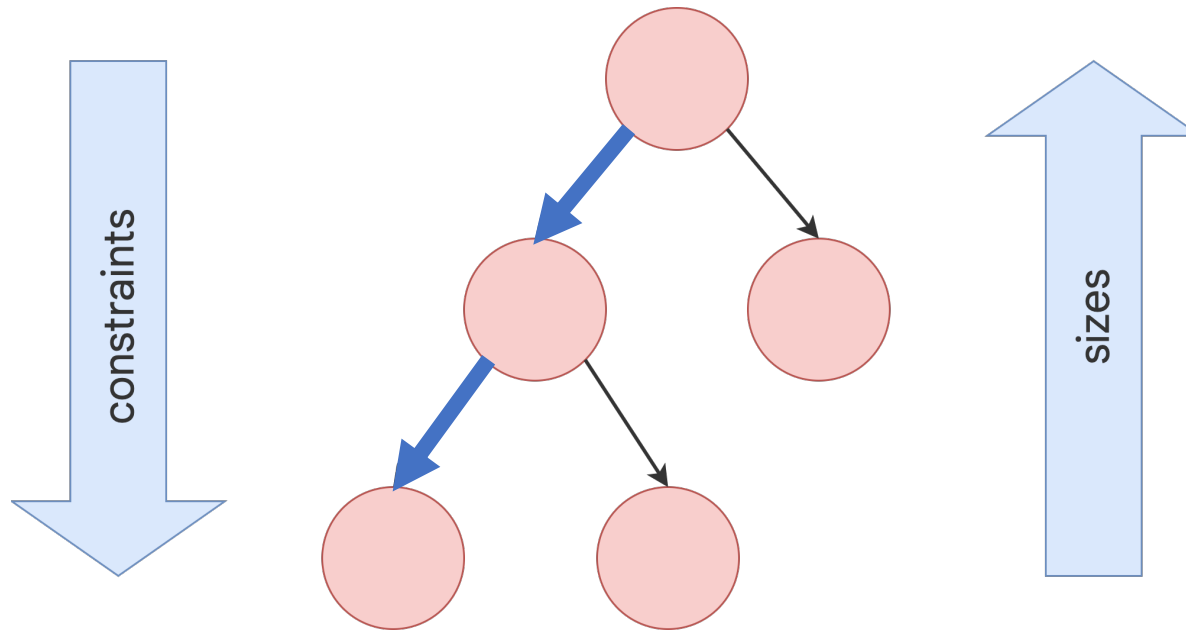
- Map element to **RenderObject** to perform actual rendering

Rendering Process



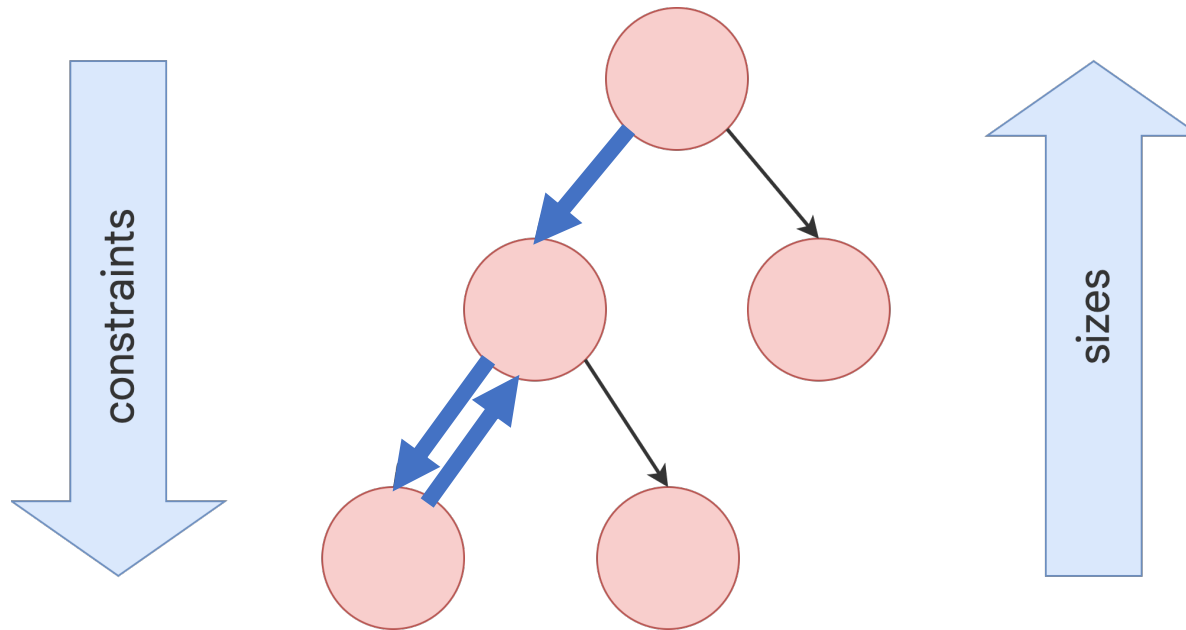
- We focus on *layout* and *painting* here

Layout: Constraints, Sizes, Positions



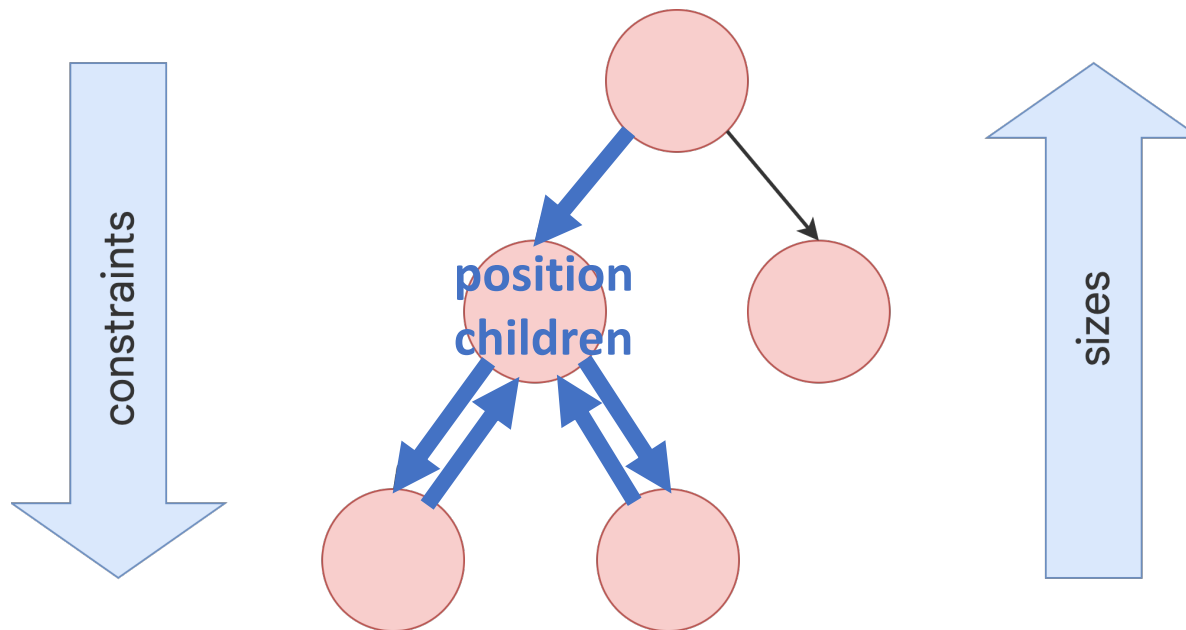
- Constraints go down (depth-first)
 - Parents tell children min/max width/height

Layout: Constraints, Sizes, Positions



- Sizes go up (depth-first)
 - Children pass sizes up to their parents

Layout: Constraints, Sizes, Positions



- Parents set positions (breadth-first)
 1. Position children based on layout concept
 2. Then, bubble up sizes to grand-parents

Constraints

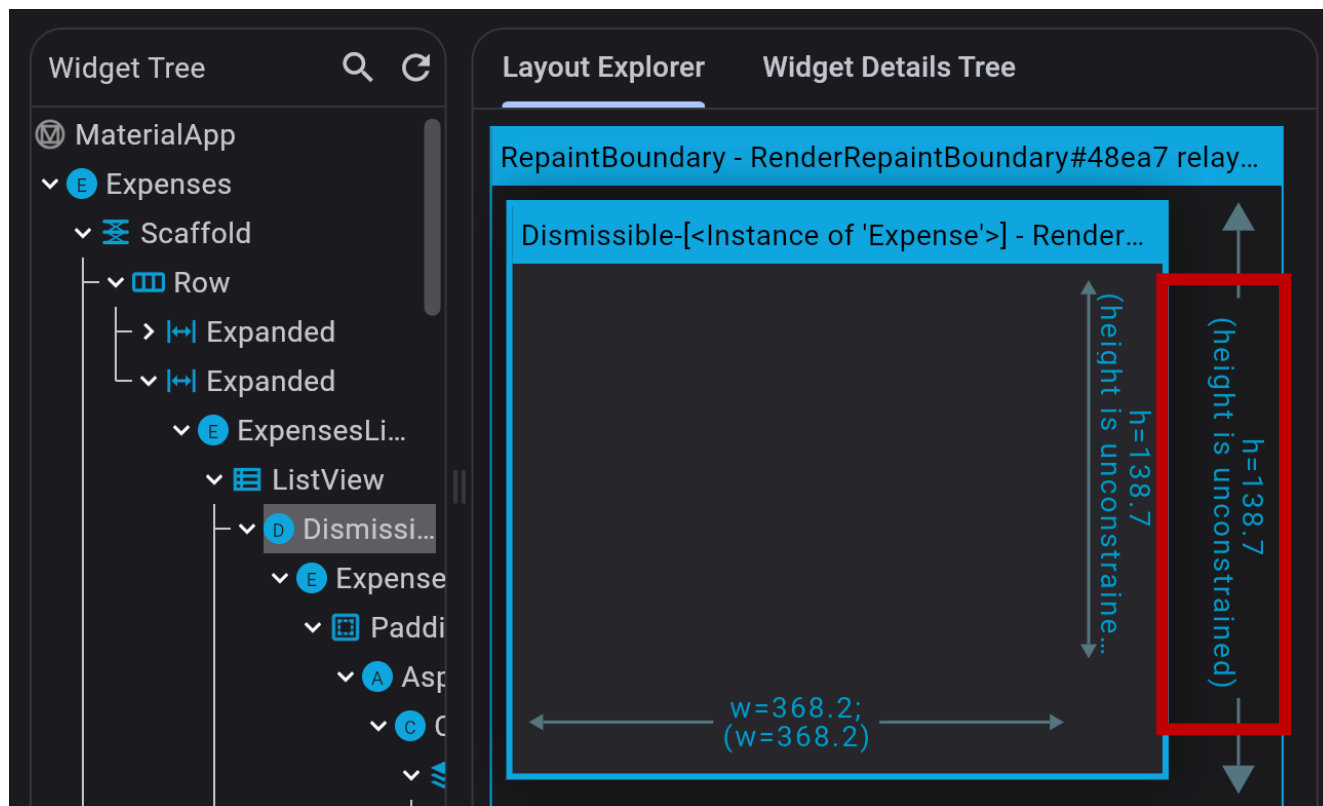
- Tight constraints (min = max)
 - `AppBar` and `Expanded` (`Flexible` with tight fit)
- Loose constraints (min < max)
 - `Center` transforms tight constraints from parent to loose constraints for its child, allowing “centering”
 - `Flexible` with loose fit
- Unbounded constraints
 - Flex box (`Row` or `Column`) unless `Flexible` presents
 - Scrollable widgets (`ListView` or [Sliver widgets](#))

Sizing

- **Be as big as possible:**
 - `Center` and `ListView`, etc.
- **Fit to children size:**
 - `Transform` and `Opacity`, etc.
- **Has intrinsic (self-determined) size:**
 - `Image` and `Text`, etc.
- **It depends:**
- **Container:** as big as possible, but fits children if it or its children has `width / height`
- **Flex box (Row or Column):** fit children if unbounded in primary direction; otherwise as big as possible
 - No nested `ListView` in former case

Inspecting Layouts

- Children of `ListView` have no constraints
- In `Row`, `Expanded` wraps `Chart` (with infinite width) to avoid layout problems



Challenge for You!

Why some widget with `width: 100` isn't 100 pixels wide?

Why that `Column` is overflowing?

What `IntrinsicWidth` is supposed to be doing?

Why some `FittedBox` isn't working?

- Try explain [layout examples](#) by yourself

Why Position Late?

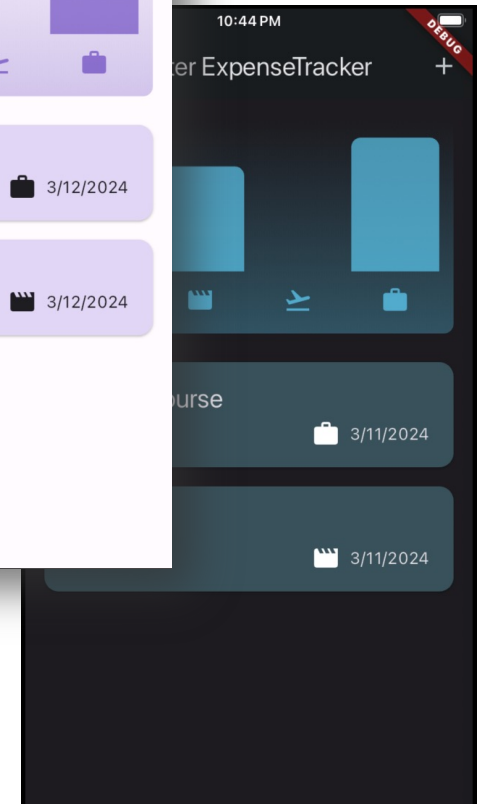
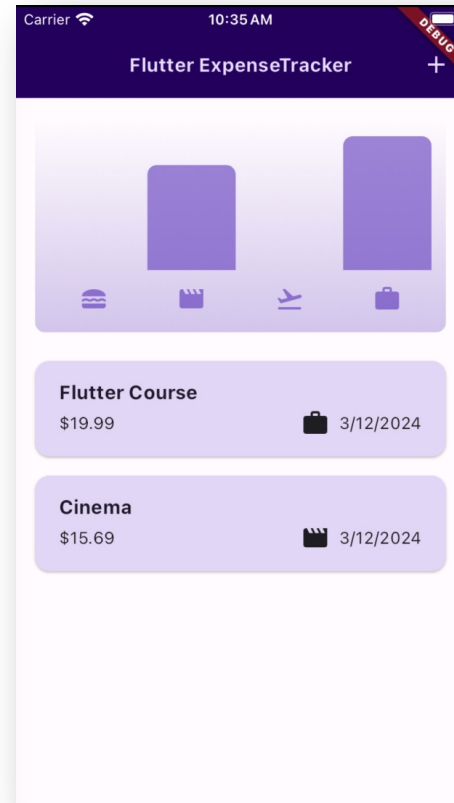
- Different from Android and iOS layout process
- Benefits:
 - Predictable & consistent widgets/ layout behavior
 - Fast: *single pass*; good for animations/transitions
 - Flexible for different screen sizes
 - Easy to understand and simplifies development process
- Example: fast scrollable [Sliver widgets](#)

Limitations

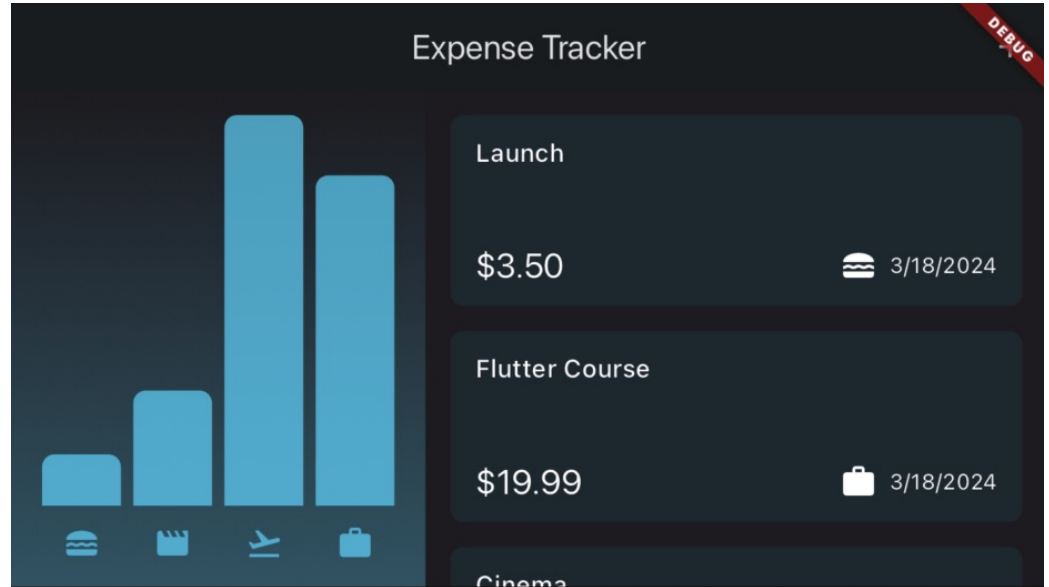
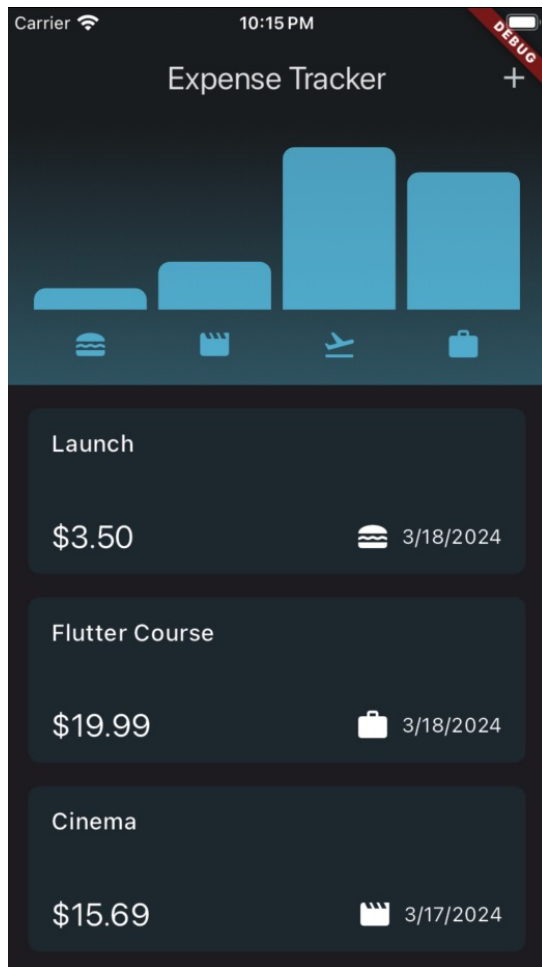
- Widget usually can't have any size it wants
 - Constraints from parent
- Widget can't know and doesn't decide its own position in the screen
 - Position determined only after layout of entire tree
- Be specific when defining alignment
 - Otherwise, some children's sizes may be ignored

Today's Topics

- Build & rendering
- Element tree
- Render tree
- **Responsive layout**
- Custom painting & parallax scrolling



Making Expense App Responsive



Locking Device Orientation

```
import 'package:flutter/services.dart';

void main() async {
  // Ensure Flutter framework is initialized
  WidgetsFlutterBinding.ensureInitialized();

  await SystemChrome.setPreferredOrientations([
    DeviceOrientation.portraitUp,
]);

  ...
}
```

- For some apps, this could be the best solution

Dynamic Layout

- At top of widget tree, use `MediaQuery` to trigger layout change
 - E.g., `Column` → `Row` when screen width > 600
 - See [predefined layout breakpoints](#)
- At lower part of tree, use `LayoutBuilder`
 - ***Get constraints from parent*** programmably
 - Then, build widget dynamically
 - See `new_expense.dart`

Handling Keyboard Inset (1/2)

- Dynamic sizing:
- With Scaffold
 - `resizeToAvoidBottomInset` set to `true` by default
 - No further action needed
- Without Scaffold (e.g., in modal)
 - Use `MediaQuery.viewInsetsOf(context).bottom` to detect KB
 - Add bottom padding dynamically

Handling Keyboard Inset (2/2)

- Avoiding obscuration:
- With Scaffold
 - Auto-scrolling available
 - But need conditional padding for `body` to create space between input widget (`TextField`) and KB
- Without Scaffold (e.g., in modal)
 - Wrap root widget with `SingleChildScrollView`
 - Implement manual scrolling
 - See `new_expense.dart`

Safe Area

- Different across devices



- Use `MediaQuery.paddingOf(context).top/bottom`
- Or `SafeArea` widget
- For modal, set `useSafeArea` argument to `true` when calling `showModalBottomSheet()`

Responsiveness ≠ Adaptiveness

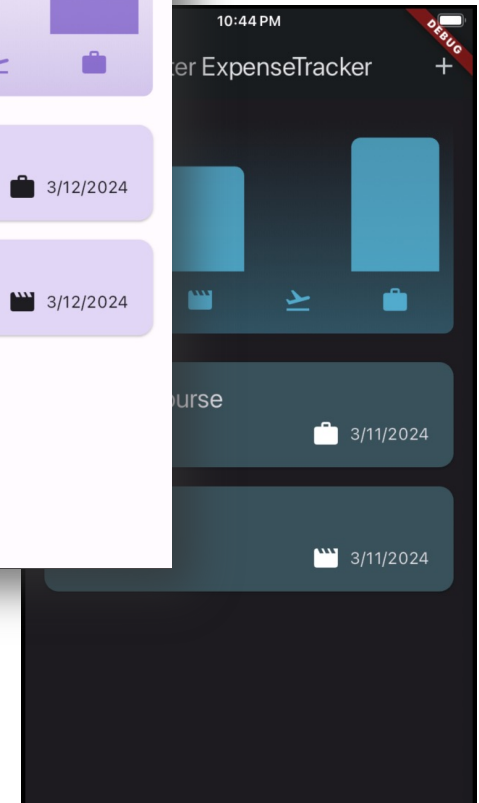
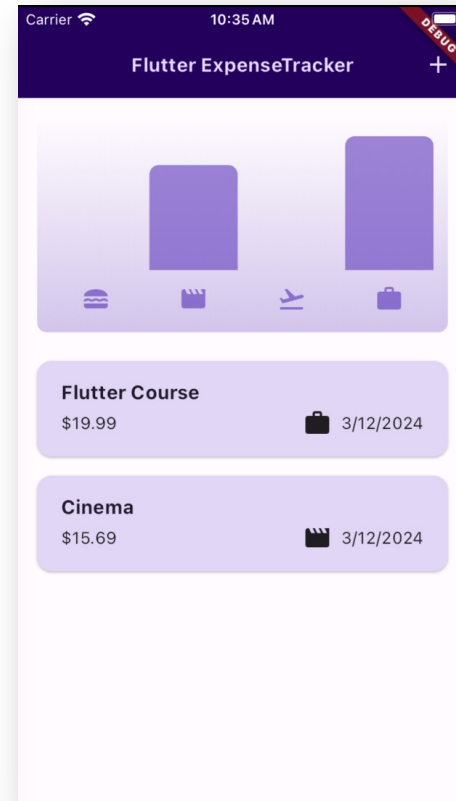
- ***Responsiveness***: UI renders well across different screen sizes and orientations
- ***Adaptiveness***: different layouts and functionalities for different platforms
- Example iOS adaptation:
 - `Platform.isIOS`
 - `showCupertinoAlert()` in `new_expenses.dart`

Automatic Platform Adaptation

- Theming
 - On iOS, title in `AppBar` is centered by default
- Platform-specific (Cupertino) widgets
- Adaptive constructors
 - E.g., `Icon.adaptive.share`, `AdaptiveDialog`

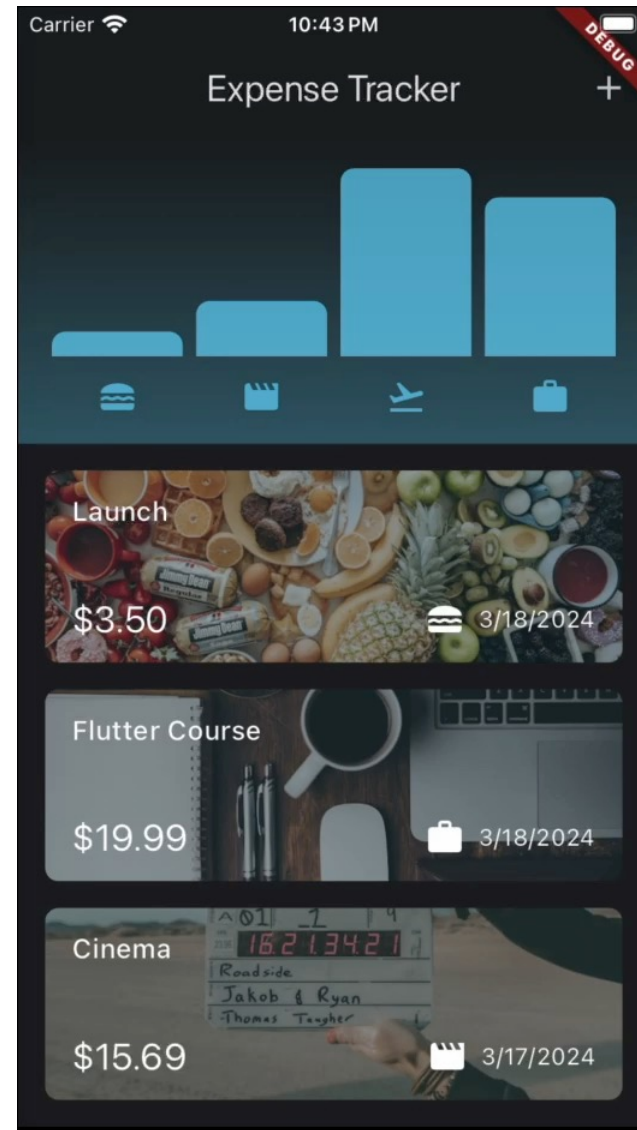
Today's Topics

- Build & rendering
- Element tree
- Render tree
- Responsive layout
- Custom painting & parallax scrolling

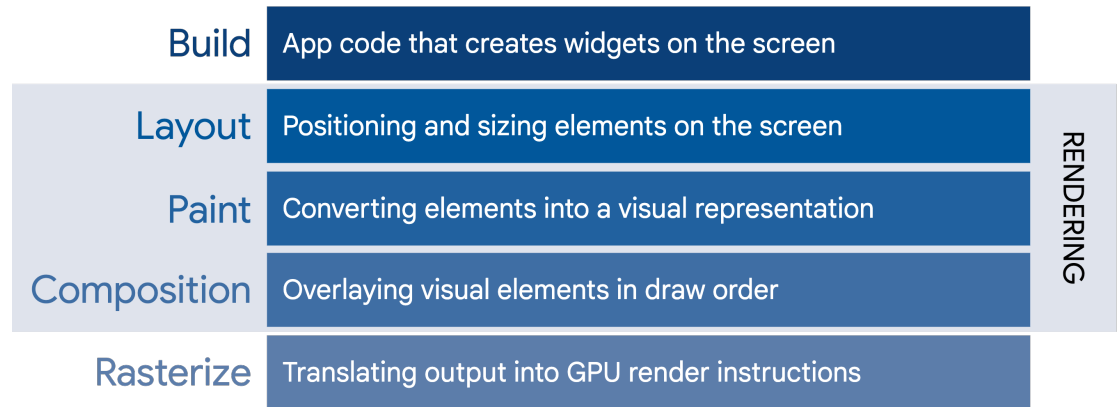


Demo: Efficient Parallax Scrolling

- No rebuild & re-layout costs while scrolling



Painting



- In painting phase, we can still apply ***matrix transformations*** to `RenderObject`
 - Translation, rotation, scaling, skewing, etc.
- [Transform](#): for single widget
- [Flow](#): for multiple widgets or custom layout

Flow Widget

- Does not pass constraints down to children
 - So, children retain their intrinsic sizes
- `FlowDelegate` offers you control on:
 - **Layout:** `getSize()` and `setChildParentData()`
 - **Paint:** `paintChildren()` and `shouldRepaint()`

Suggested Reading

- Take [layout examples](#) as a challenge
- [Transform](#) widget

- Sliver Widgets
 - [Short intro](#) (CustomScrollView + SliverAppBar)
 - [Longer hands-on tutorial](#)
- [Adaptive design](#) and [automation](#)
- [Flutter Internals](#)*