Lab session 2: UI Design and Creating Flutter Apps

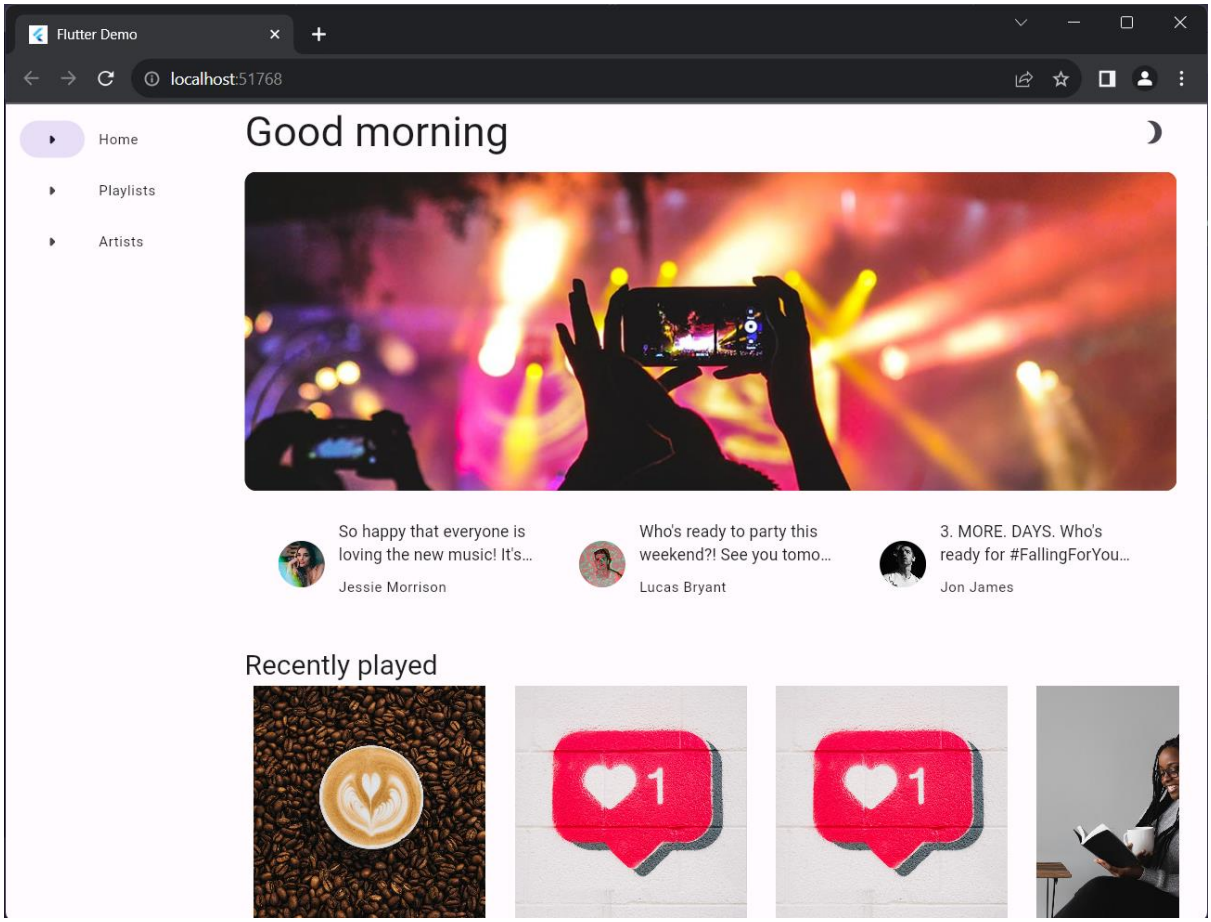
Task-2 Take your Flutter app from boring to beautiful

In this codelab, we will enhance a Flutter music application, taking it from boring to beautiful. To accomplish this, this codelab uses tools and APIs introduced in Material 3.

Get the codelab starter app

We will clone a starter app from Github as a base for our application.

Initially, the app's design looks like this:



We will walk through the steps we took to implement a more visual appealing and user-friendly UI.

Take advantage of typography

How text is presented shapes a user's first impression of the app.

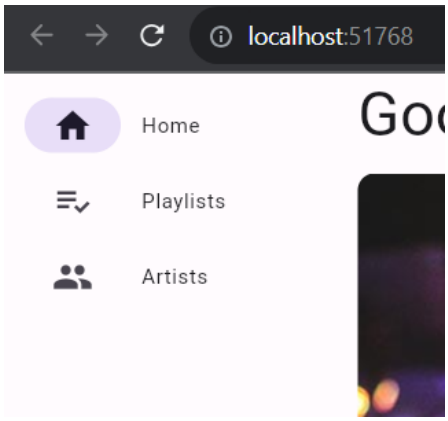
Show, don't tell

Wherever possible, "show" instead of "tell". It's better to illustrate texts with some visual icons instead of pushing users to read before understanding the use of a button, or any tab.

1. In lib/src/shared/router.dart, add distinct leading icons for each navigation destination (home, playlist, and people):

```
lib > src > shared > router.dart > destinations
22  const List<NavigationDestination> destinations = [
23    NavigationDestination(
24      label: 'Home',
25      icon: Icon(Icons.home), // Modify this line
26      route: '/',
27    ), // NavigationDestination
28    NavigationDestination(
29      label: 'Playlists',
30      icon: Icon(Icons.playlist_add_check), // Modify
31      route: '/playlists',
32    ), // NavigationDestination
33    NavigationDestination(
34      label: 'Artists',
35      icon: Icon(Icons.people), // Modify this line
36      route: '/artists',
37    ), // NavigationDestination
38  ];
```

Result:



Choose fonts thoughtfully

Fonts set the personality of an application, so choosing the right font is crucial.

With this in mind, head over to Google Fonts and choose a sans-serif font, like Montserrat, since the music app is intended to be playful and fun.

1. From the command line, pull in the google_fonts package. This also updates the pubspec file to add the fonts as an app dependency.

```
PS D:\Dev\flutter-codelabs\boring_to_beautiful\step_01> flutter pub add google_fonts
Resolving dependencies...
  _fe_analyzer_shared 64.0.0 (67.0.0 available)
  analyzer 6.2.0 (6.4.1 available)
+ ffi 2.1.0 (2.1.2 available)
+ google_fonts 6.1.0
+ http 1.2.0
  matcher 0.12.16 (0.12.16+1 available)
  material_color_utilities 0.5.0 (0.8.0 available)
  meta 1.10.0 (1.11.0 available)
```

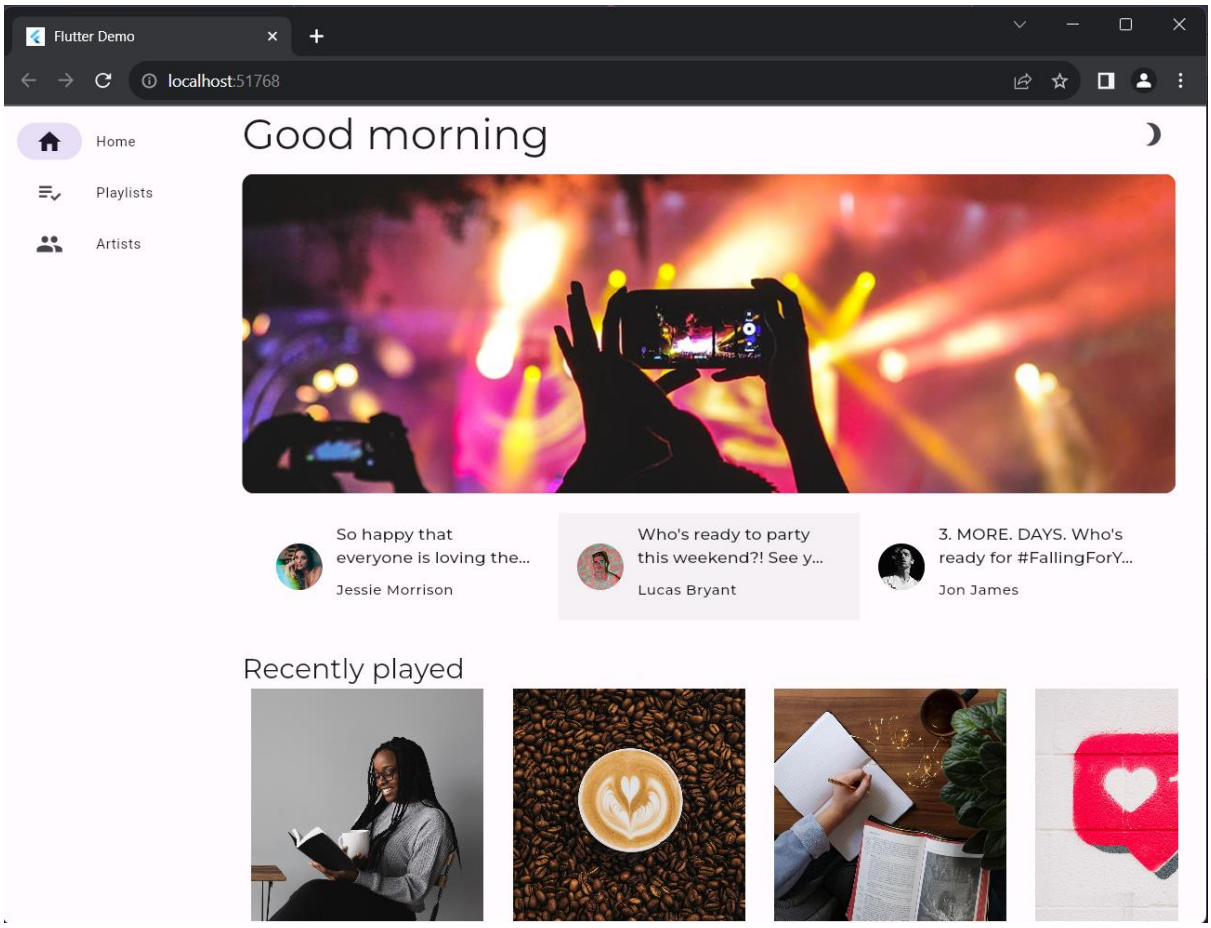
2. In lib/src/shared/extensions.dart, import the new package:

```
lib > src > shared > extensions.dart > ...
4
5   import 'package:flutter/material.dart';
6   // Add Google Fonts Package import
7   import 'package:google_fonts/google_fonts.dart';
```

3. Set the Montserrat TextTheme:

```
lib > src > shared > extensions.dart > {} TypographyUtils > textTheme
4
5   import 'package:flutter/material.dart';
6   // Add Google Fonts Package import
7   import 'package:google_fonts/google_fonts.dart';
8
9   extension TypographyUtils on BuildContext {
10    ThemeData get theme => Theme.of(this);
11    TextTheme get textTheme => GoogleFonts.montserratTextTheme(theme.textTheme);
```

4. Hot reload to activate the changes. (Use the button in your IDE or, from the command line, enter r to hot reload.):



We can observe that the font as changed, giving a new look the app.

Set the theme

Themes help bring a structured design and uniformity to an app by specifying a set system of colors and text styles. Themes enable to quickly implement a UI without having to stress over minor details like specifying the exact color for every single widget.

This example uses a theme provider located in lib/src/shared/providers/theme.dart to create consistently-themed widgets and colors throughout the app:

lib > src > shared > providers > theme.dart > ...

```dart

import 'dart:math';

import 'package:flutter/material.dart';
import 'package:material_color_utilities/material_color_utilities.dart';

class NoAnimationPageTransitionsBuilder extends PageTransitionsBuilder {
  const NoAnimationPageTransitionsBuilder();

  @override
  Widget buildTransitions<T>(
    PageRoute<T> route,
    BuildContext context,
    Animation<double> animation,
    Animation<double> secondaryAnimation,
    Widget child,
  ) {
    return child;
  }
}

class ThemeSettingChange extends Notification {
  ThemeSettingChange({required this.settings});
  final ThemeSettings settings;
}
```

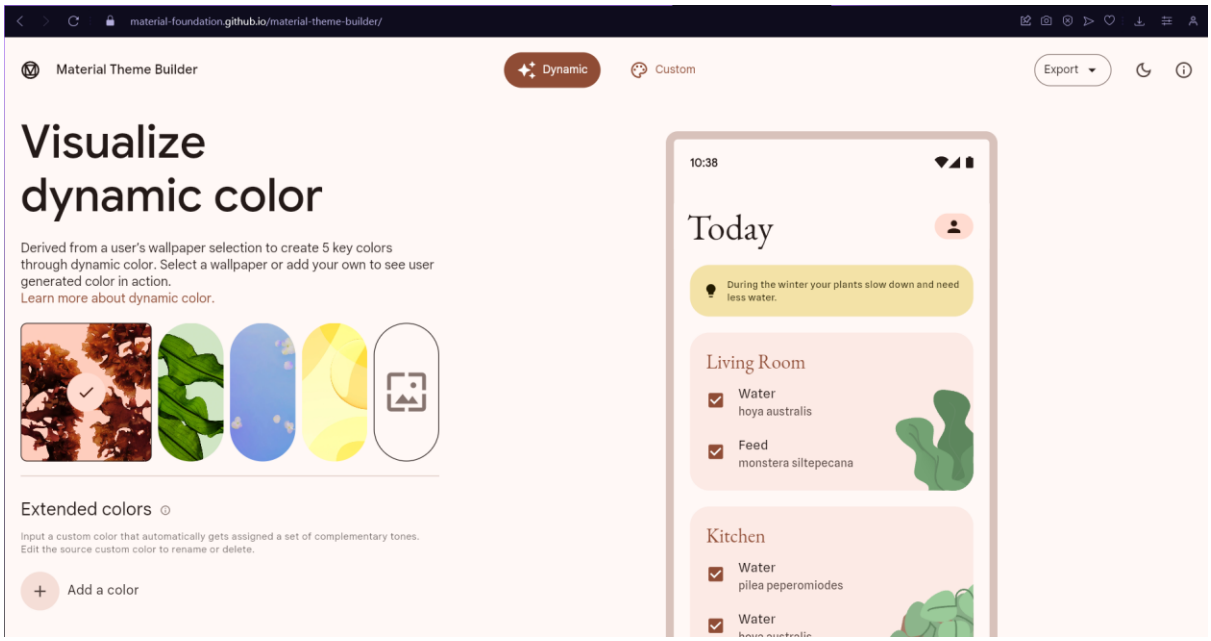
1. To use the provider, create an instance and pass it to the scoped theme object in MaterialApp, located in lib/src/shared/app.dart. It will be inherited by any nested Theme objects.

lib > src > shared > app.dart > _MyAppState > build

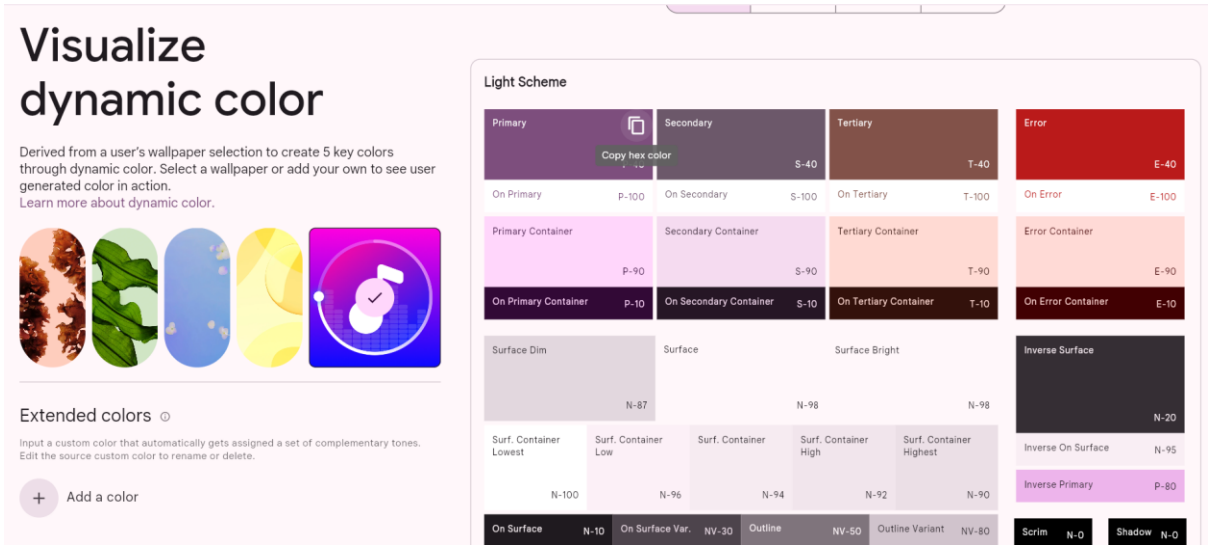
```dart
            return true;
          },
          child: ValueListenableBuilder<ThemeSettings>(
            valueListenable: settings,
            builder: (context, value, _) {
              // Create theme instance
              final theme = ThemeProvider.of(context);
              return MaterialApp.router(
                debugShowCheckedModeBanner: false,
                title: 'Flutter Demo',
                // Add theme
                theme: theme.light(settings.value.sourceColor),
```

Now that the theme is set up, choose colors for the application.

2. To choose a source color for the application, open the Material Theme Builder and explore different colors for the UI. It's important to select a color that fits the brand aesthetic and/or your personal preference.



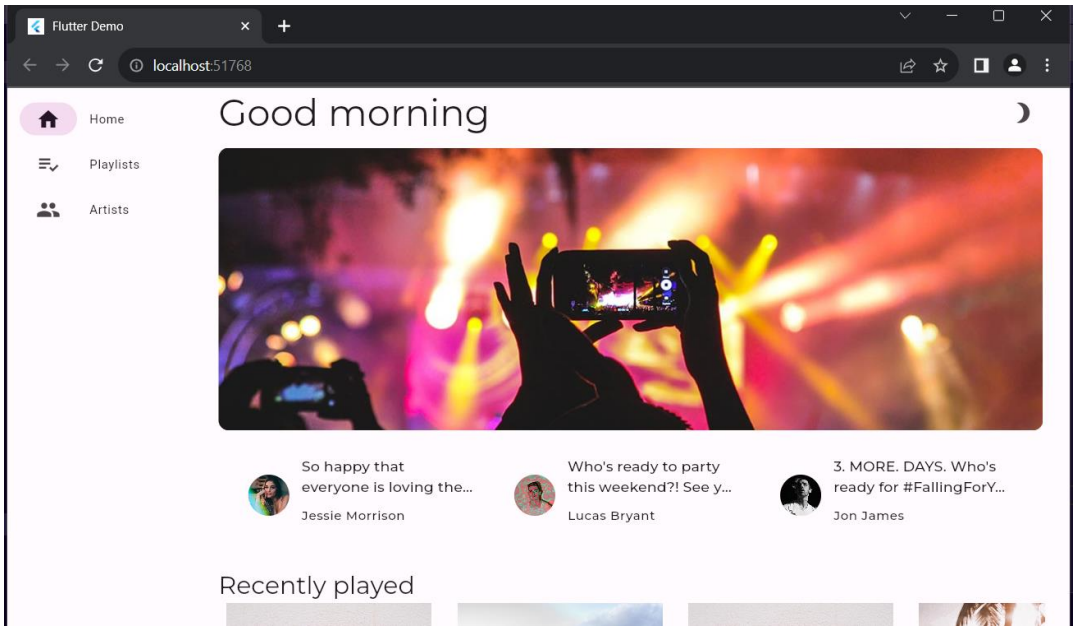
3. After creating a theme, right-click the Primary color bubble—this opens a dialog containing the hex value of the primary color. Copy this value.



4. Pass the primary color's hex value to the theme provider. For example, the hex color #00cbe6 is specified as Color(0xff00cbe6). The ThemeProvider generates a ThemeData that contains the set of complementary colors that you previewed in Material Theme Builder:

```
lib > src > shared > app.dart > _MyAppState > settings
19  }
20
21  class _MyAppState extends State<MyApp> {
22    final settings = ValueNotifier(ThemeSettings(
23      sourceColor: const Color(0x7d4e7d), // Replace this color
24      themeMode: ThemeMode.system,
25    )); // ThemeSettings // ValueNotifier
```

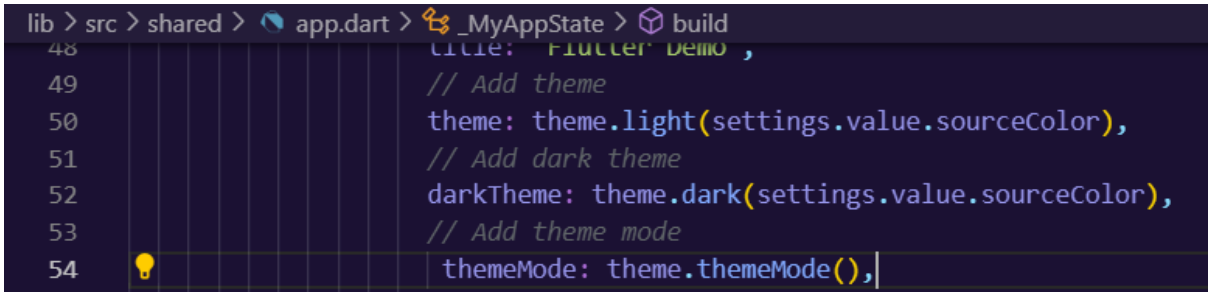
5. Hot restart the app. With the primary color in place, the app starts to feel more expressive:



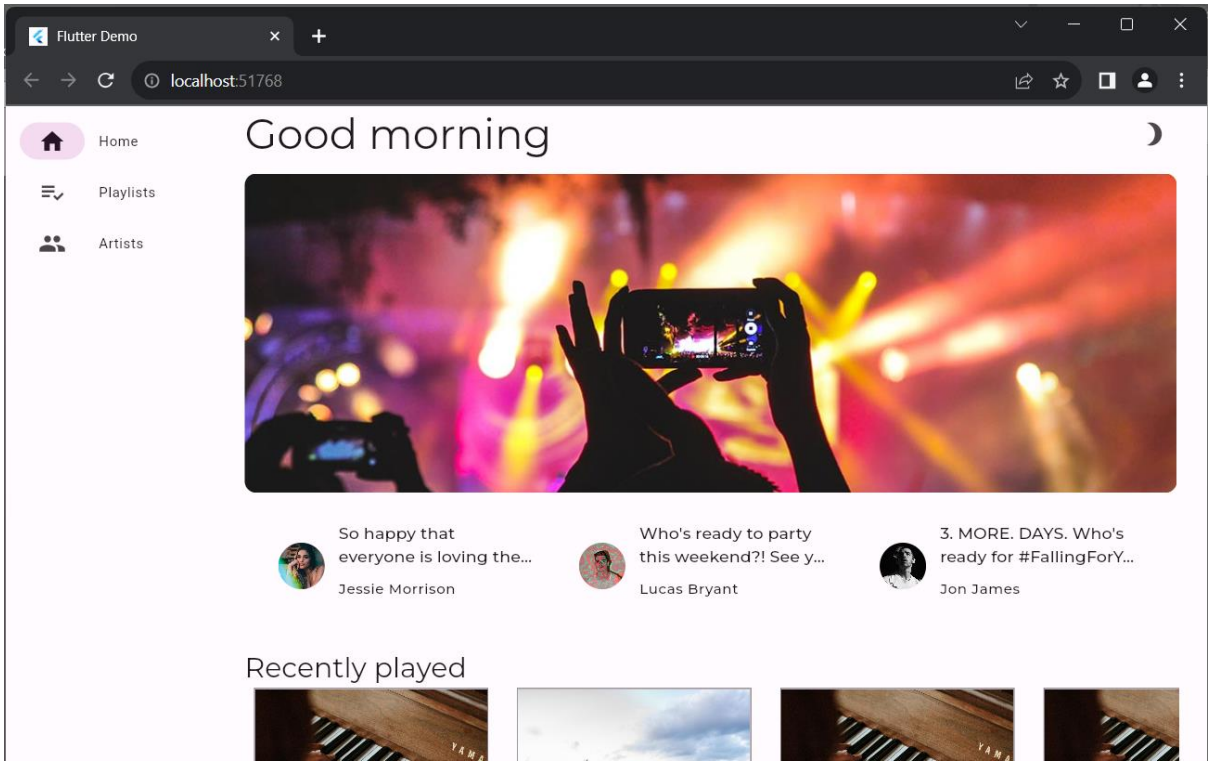
6. To use a particular color, access a color role on the colorScheme. Go to lib/src/shared/views/outlined_card.dart and give the OutlinedCard a border:

```
lib > src > shared > views > outlined_card.dart > _OutlinedCardState > build
22    @override
23    Widget build(BuildContext context) {
24      return MouseRegion(
25        cursor: widget.clickable
26            ? SystemMouseCursors.click
27            : SystemMouseCursors.basic,
28        child: Container(
29          // Add box decoration here
30          decoration: BoxDecoration(
31            border: Border.all(
32              color: Theme.of(context).colorScheme.outline,
33              width: 1,
34            ), // Border.all
35          ), // BoxDecoration
36          child: widget.child,
37        ), // Container
38      ); // MouseRegion
39    }
40  }
```

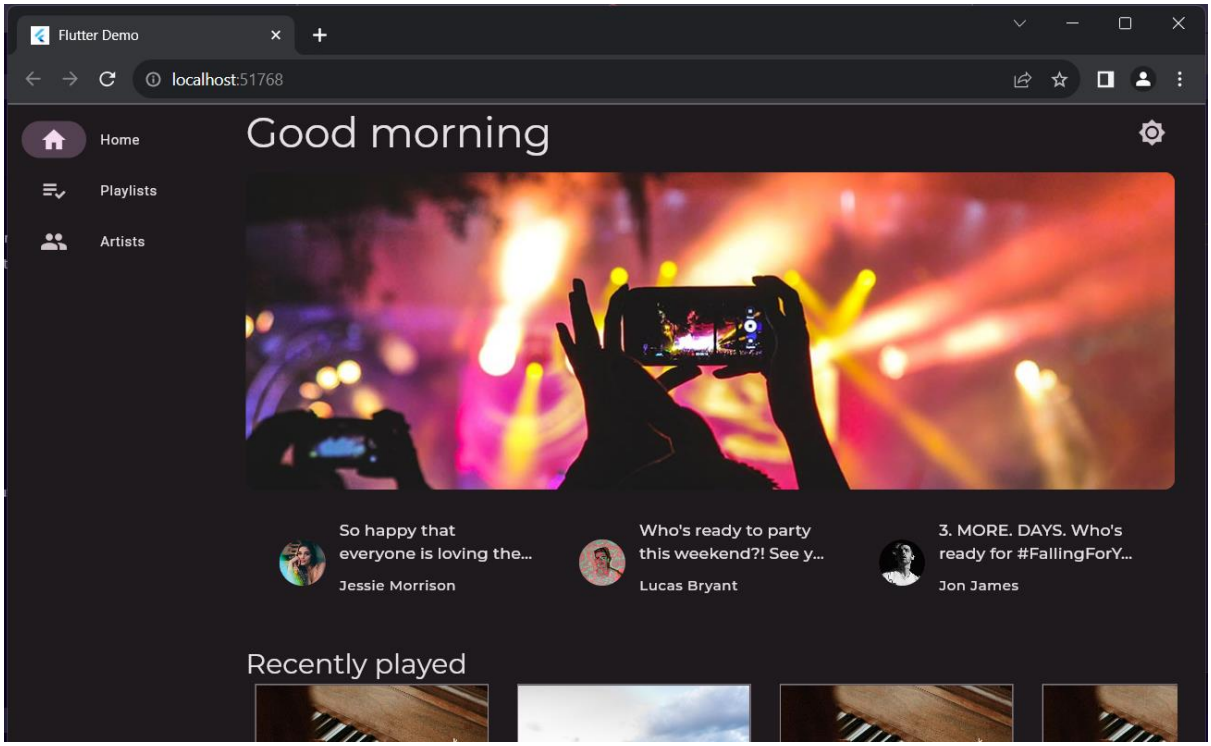
7. The user can set the app brightness in the device's system settings. In lib/src/shared/app.dart, when the device is set to dark mode, return a dark theme and theme mode to the MaterialApp.



Theme Mode:



Click the moon icon in the top right corner to enable dark mode:



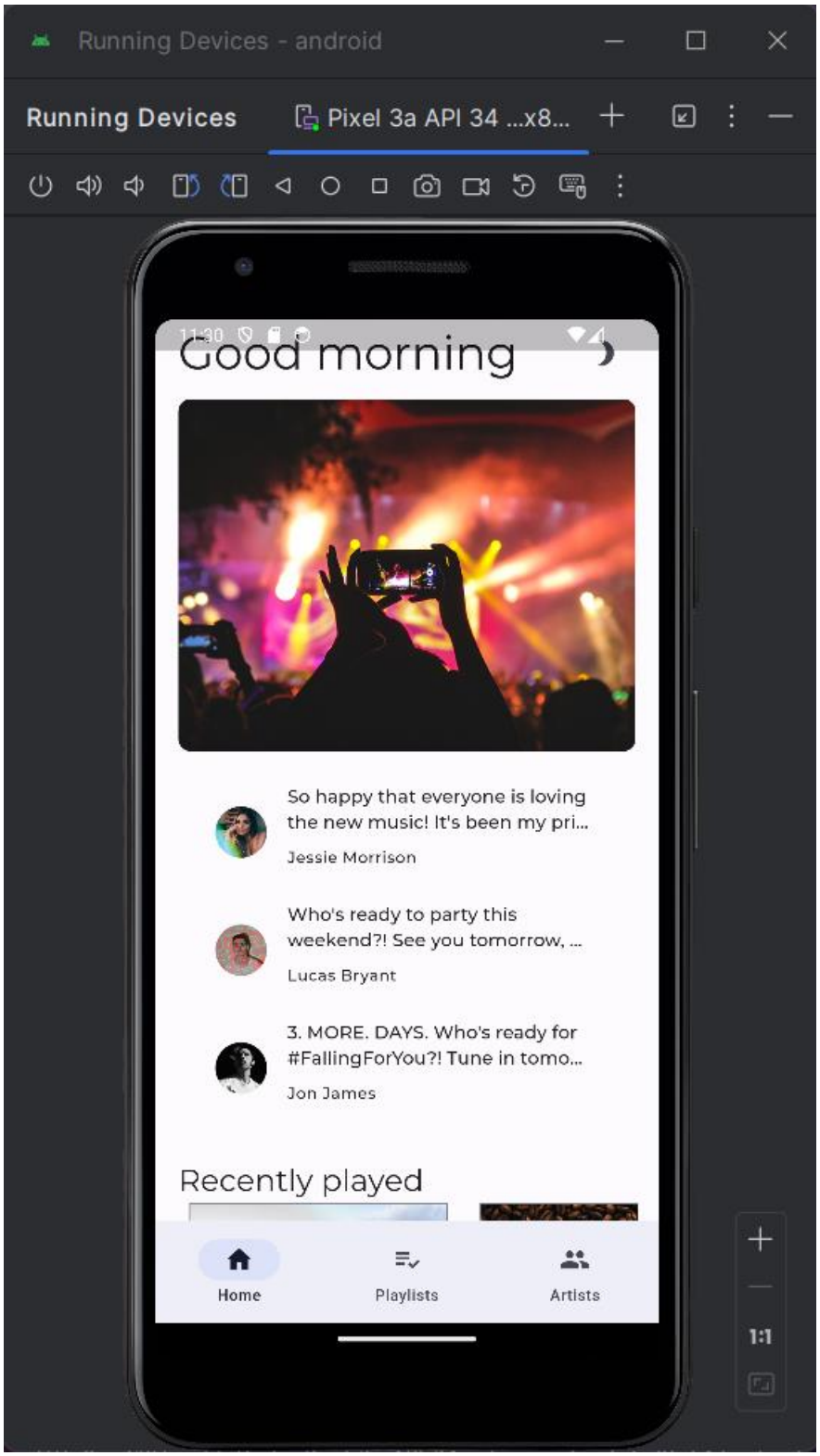
We can observe with the home Tab Icon that the colors adapts itself with the theme.

Add adaptive design

With Flutter, we can build apps that run almost anywhere, but that's not to say that every app is expected to behave the same everywhere. Users have come to expect different behaviors and features from different platforms.

1. The lib/src/shared/views/adaptive_navigation.dart file contains a navigation class where we can provide a list of destinations and content to render the body. Since weuse this layout on multiple screens, there's a shared base layout to pass into each child. Navigation rails are good for desktop and large screens, but make the layout mobile friendly by showing a bottom navigation bar on mobile instead.

Result:



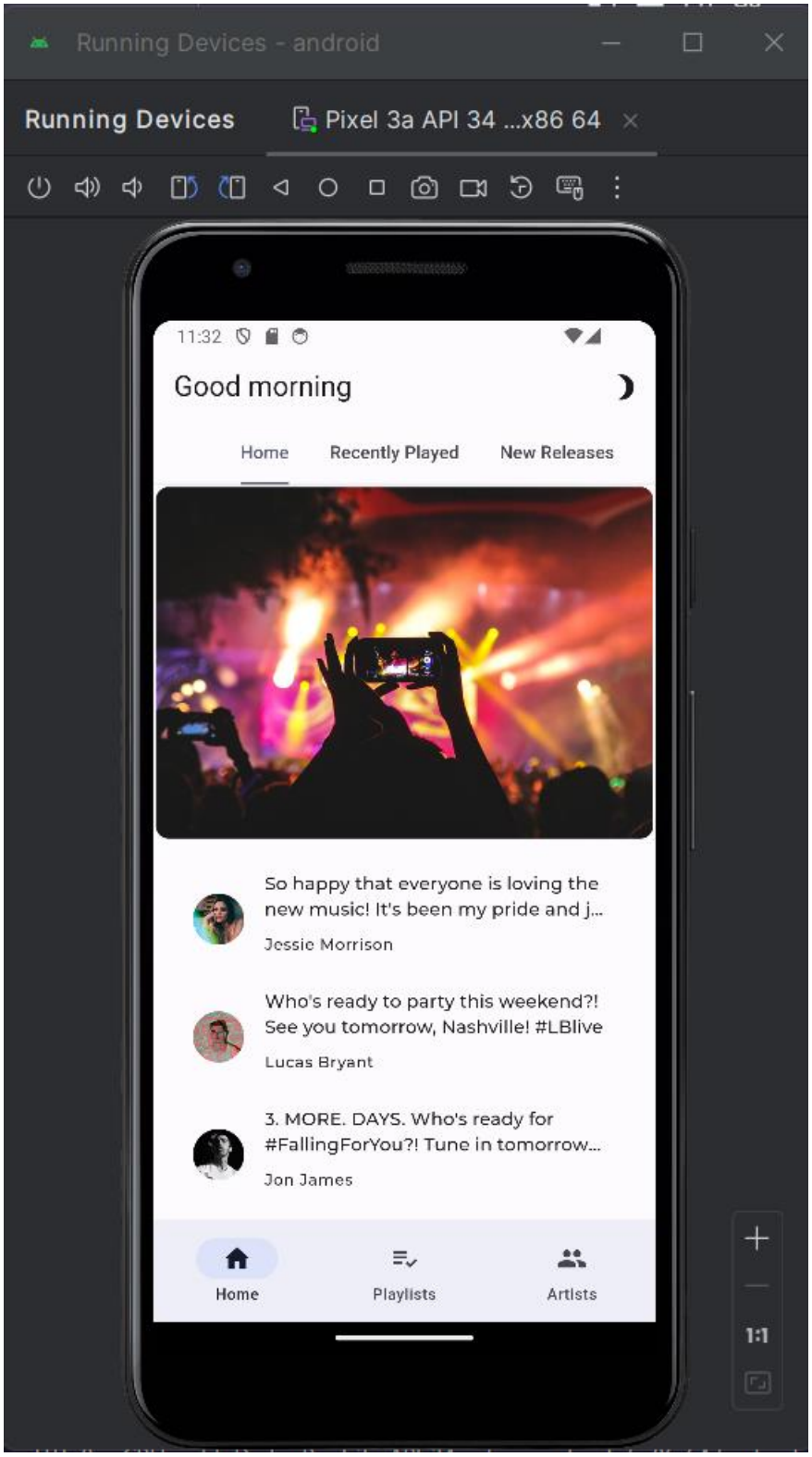
Not all screens are the same size. If we tried to display the desktop version of the app on our phone, we would have to do some combination of squinting and zooming to see everything. We want our app to change how it looks based on the screen where it's displayed. With responsive design, we ensure that the app looks great on screens of all sizes.

2. An adaptive layout needs two layouts: one for mobile, and a responsive layout for larger screens. The LayoutBuilder currently returns only a desktop layout. In lib/src/features/home/view/home_screen.dart build the mobile layout as a TabBar and TabBarView with 4 tabs.

lib > src > features > home > view > home_screen.dart > _HomeScreenState > build

```
30      final List<Artist> artists = artistsProvider.artists;
31      return LayoutBuilder(
32        builder: (context, constraints) {
33          // Add conditional mobile layout
34          if (constraints.isMobile) {
35            return DefaultTabController(
36              length: 4,
37              child: Scaffold(
38                appBar: AppBar(
39                  centerTitle: false,
40                  title: const Text('Good morning'),
41                  actions: const [BrightnessToggle()],
42                  bottom: const TabBar(
43                    isScrollable: true,
44                    tabs: [
45                      Tab(text: 'Home'),
46                      Tab(text: 'Recently Played'),
47                      Tab(text: 'New Releases'),
48                      Tab(text: 'Top Songs'),
49                    ],
50                  ), // TabBar
51                ), // AppBar
52                body: LayoutBuilder(
53                  builder: (context, constraints) => TabBarView(
54                    children: [
55                      SingleChildScrollView(
```

Result:



Now we can switch through different tabs.

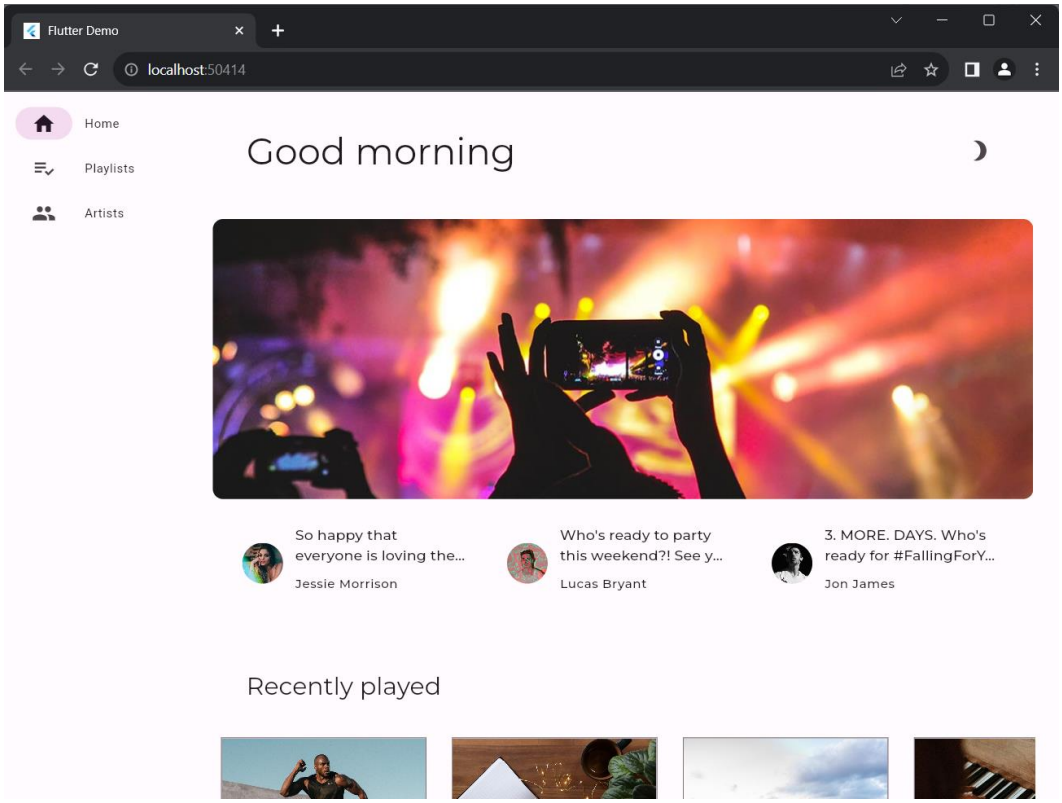
Use whitespace

Whitespace is an important visual tool for an app, creating an organizational break between sections.

Whitespace is an important visual tool for your app, creating an organizational break between sections.

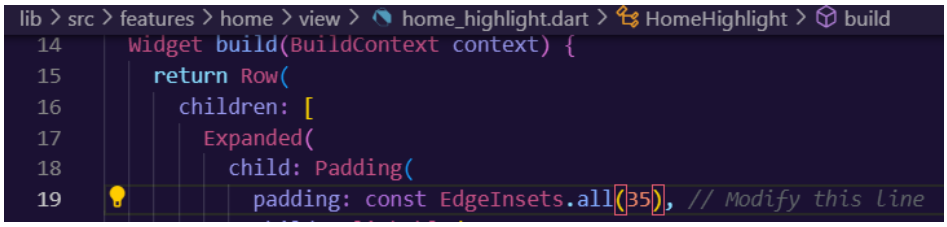
1. Wrap a widget with a Padding object to add whitespace around that widget. Increase all of the padding values currently in lib/src/features/home/view/home_screen.dart to 35:

2. Hot reload the app. It should look the same as before, but with more whitespace between the widgets.

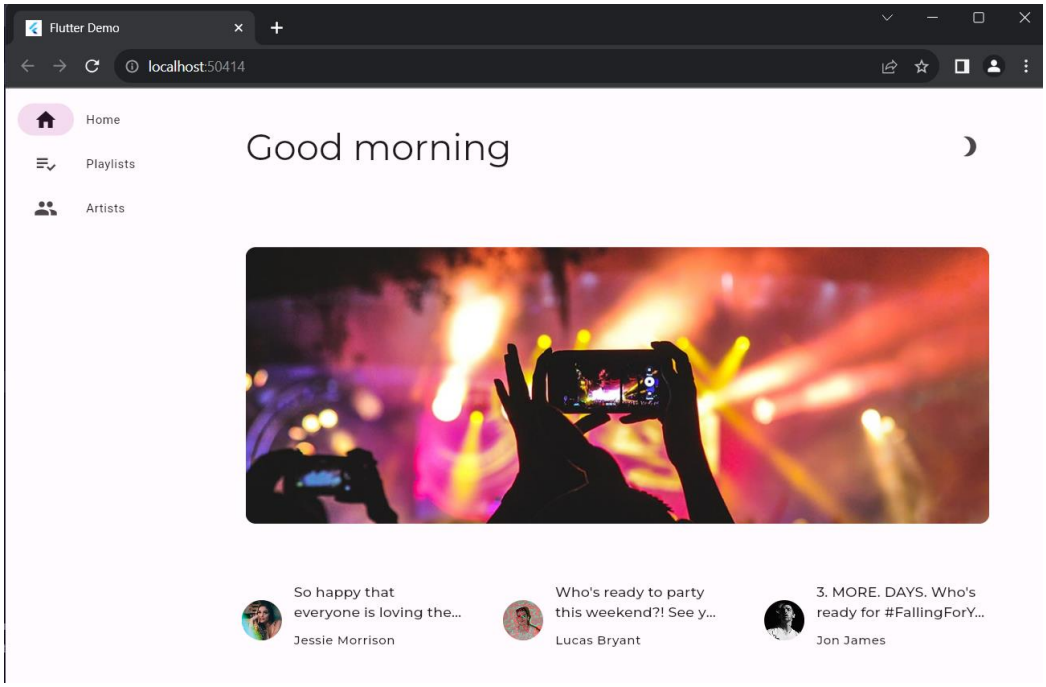


The additional padding looks better, but the highlight banner at the top is still too close to the edges.

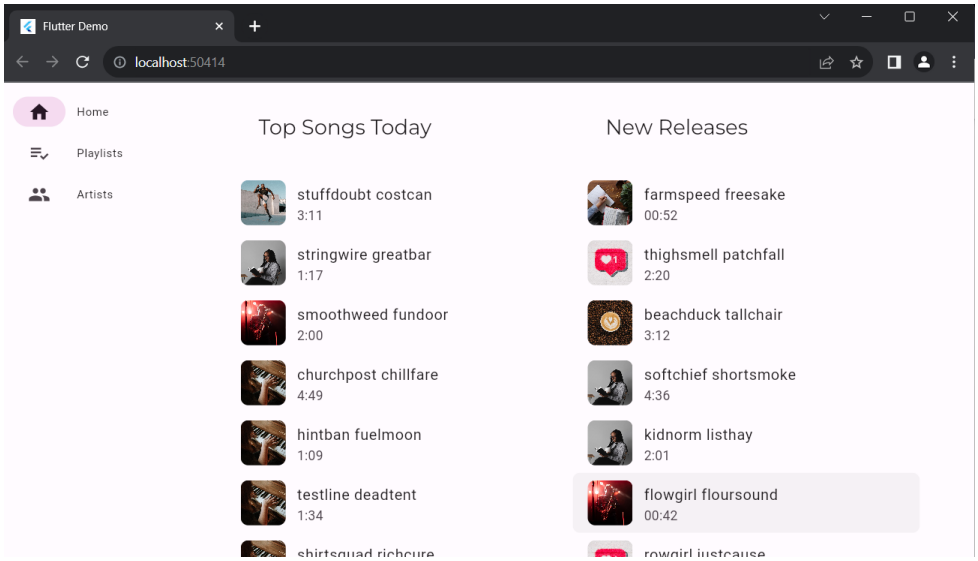
3. In lib/src/features/home/view/home_highlight.dart, change the padding on the banner to 35:



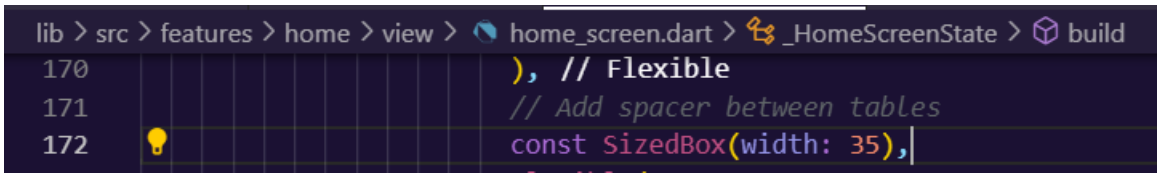
4. Hot reload the app.



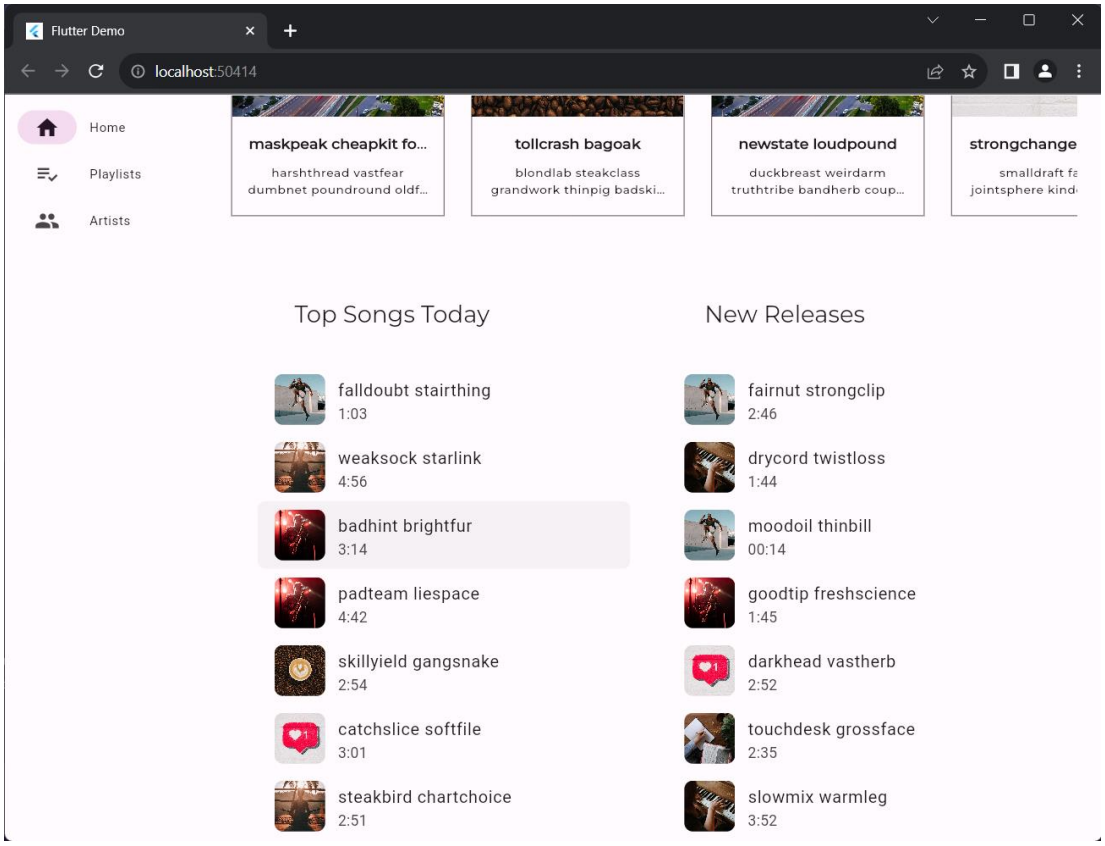
The two playlists at the bottom have no whitespace between them, so they look like they belong to the same table.



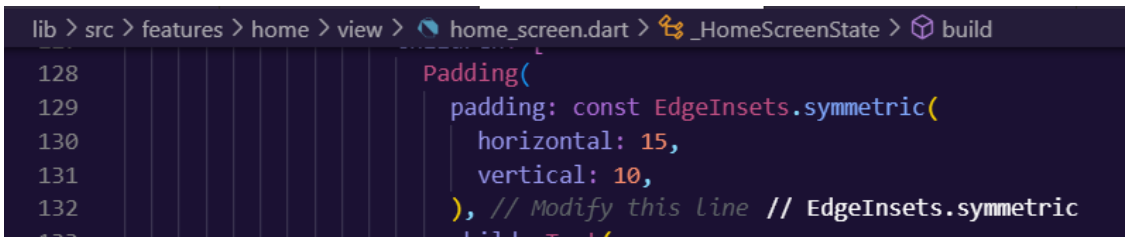
5. Add whitespace between the playlists by inserting a size widget to the Row that contains them. In lib/src/features/home/view/home_screen.dart, add a SizedBox with a width of 35:



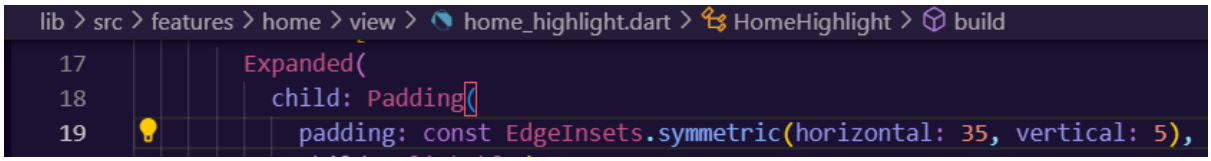
6. Hot reload the app. The app should look as follows:



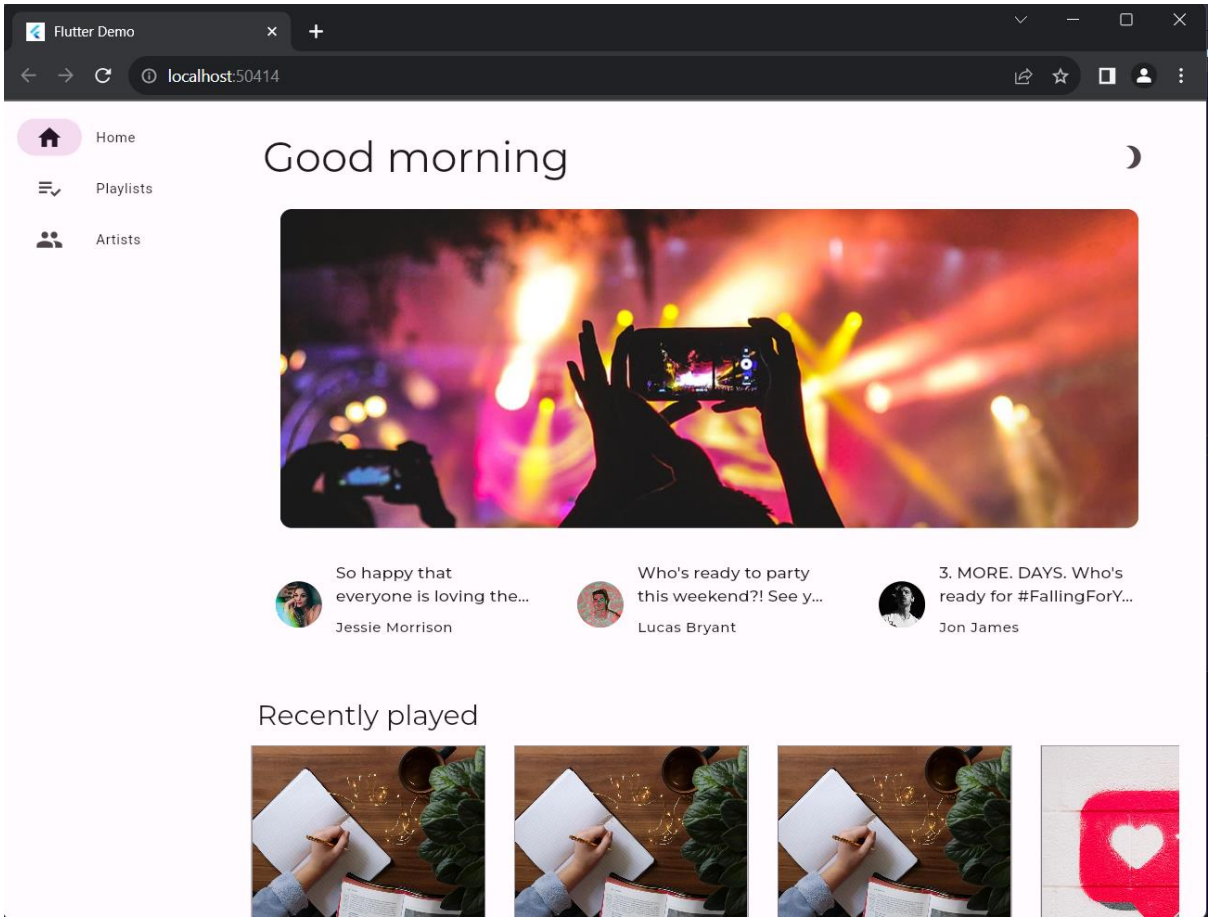
7. So far, we have set all padding (both horizontal and vertical) for the widgets on the home screen to 35 with EdgeInsets.all(35), but we can set the padding for each of the edges independently, too. Customize the padding to fit the space better.



8. In lib/src/features/home/view/home_highlight.dart, set the left and right padding on the banner to 35, and the top and bottom padding to 5:



9. Hot reload the app.



The layout and spacing look much better!

For the finishing touch, add some motion and animation.

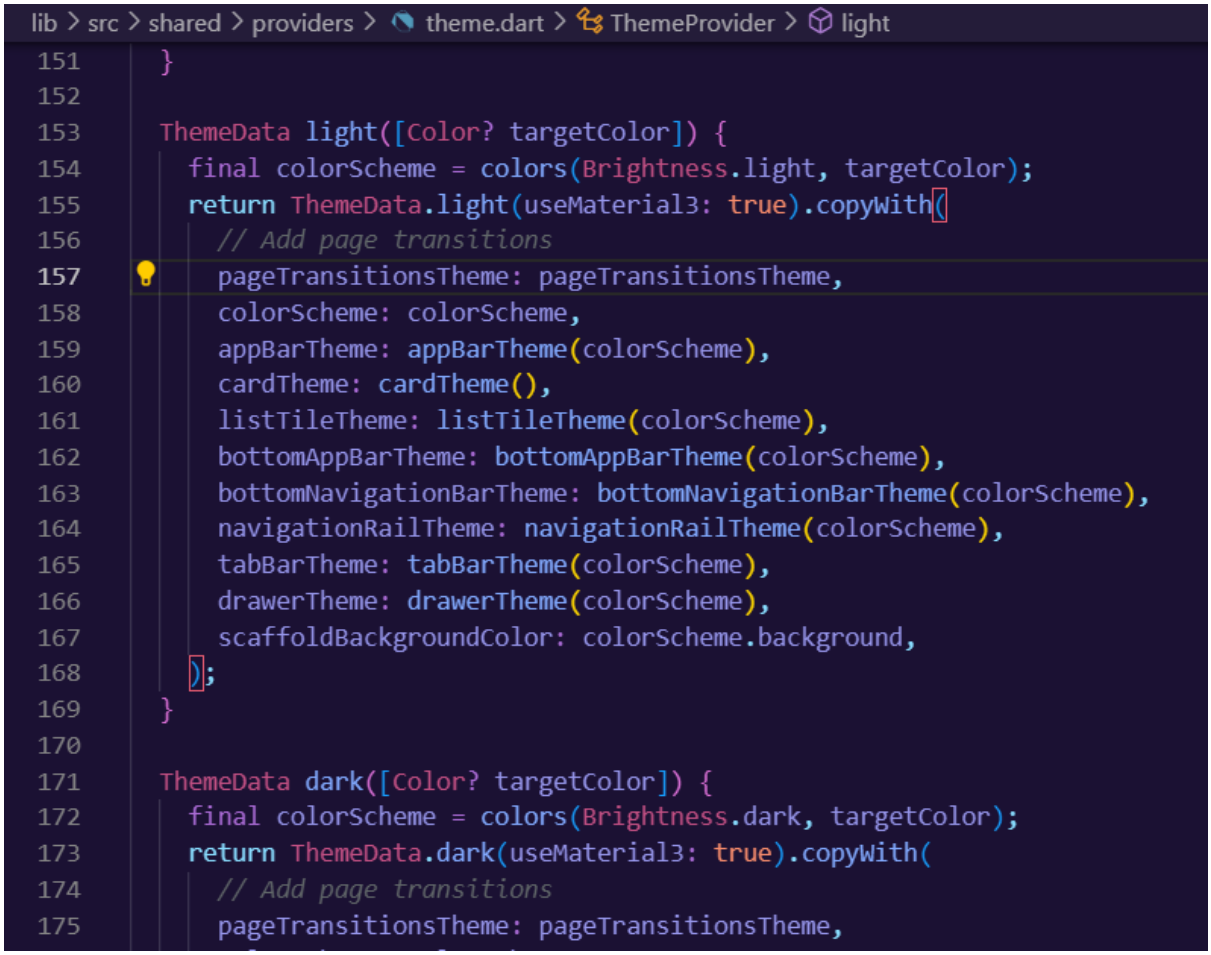
Add motion and animation

Motion and animation are great ways to introduce movement and energy, and to provide feedback when the user interacts with the app.

Animate between screens

The ThemeProvider defines a PageTransitionsTheme with screen transition animations for mobile platforms (iOS, Android). Desktop users already get feedback from their mouse or trackpad click, so a page transition animation isn't needed.

1. Pass the PageTransitionsTheme to both the light and dark themes in lib/src/shared/providers/theme.dart



Add hover states

One way to add motion to a desktop app is with hover states, where a widget changes its state (such as color, shape, or content), when the user hovers the cursor over it.

By default, the _OutlinedCardState class (used for the "recently played" playlist tiles), returns a MouseRegion—which turns the cursor arrow into a pointer on hover—but you can add more visual feedback.

1. Open lib/src/shared/views/outlined_card.dart and replace its contents with the following implementation to introduce a _hovered state.

lib > src > shared > views > outlined_card.dart > ...

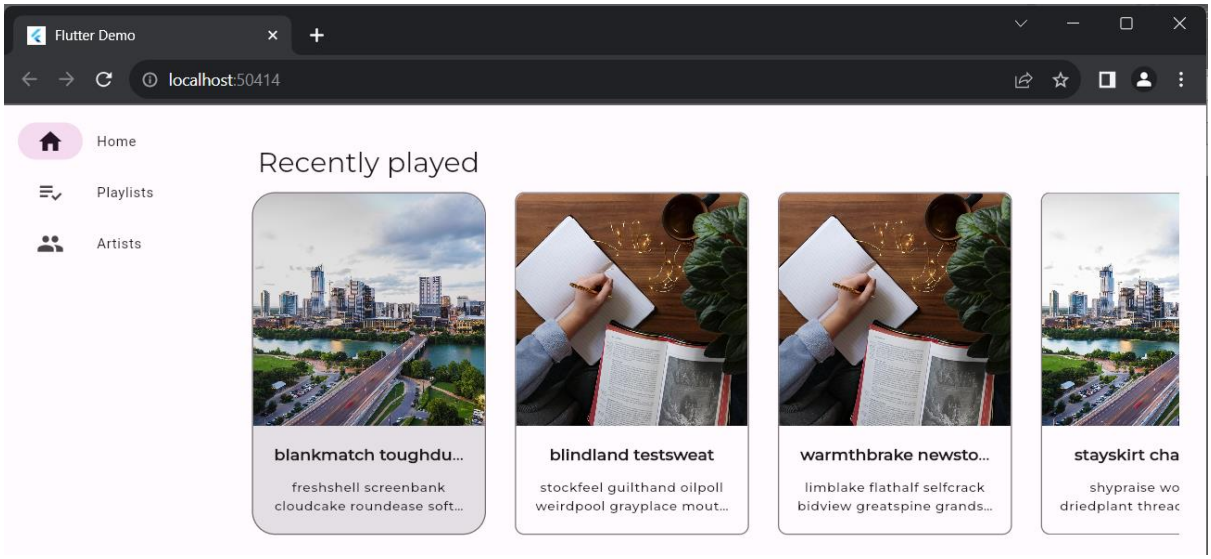
```dart
import 'package:flutter/material.dart';

class OutlinedCard extends StatefulWidget {
  const OutlinedCard({
    Key? key,
    required this.child,
    this.clickable = true,
  }) : super(key: key);
  final Widget child;
  final bool clickable;
  @override
  State<OutlinedCard> createState() => _OutlinedCardState();
}

class _OutlinedCardState extends State<OutlinedCard> {
  bool _hovered = false;

  @override
  Widget build(BuildContext context) {
    final borderRadius = BorderRadius.circular(_hovered ? 20 : 8);
    const animationCurve = Curves.easeInOut;
    return MouseRegion(
      onEnter: (_) {
        if (!widget.clickable) return;
        setState(() {
```

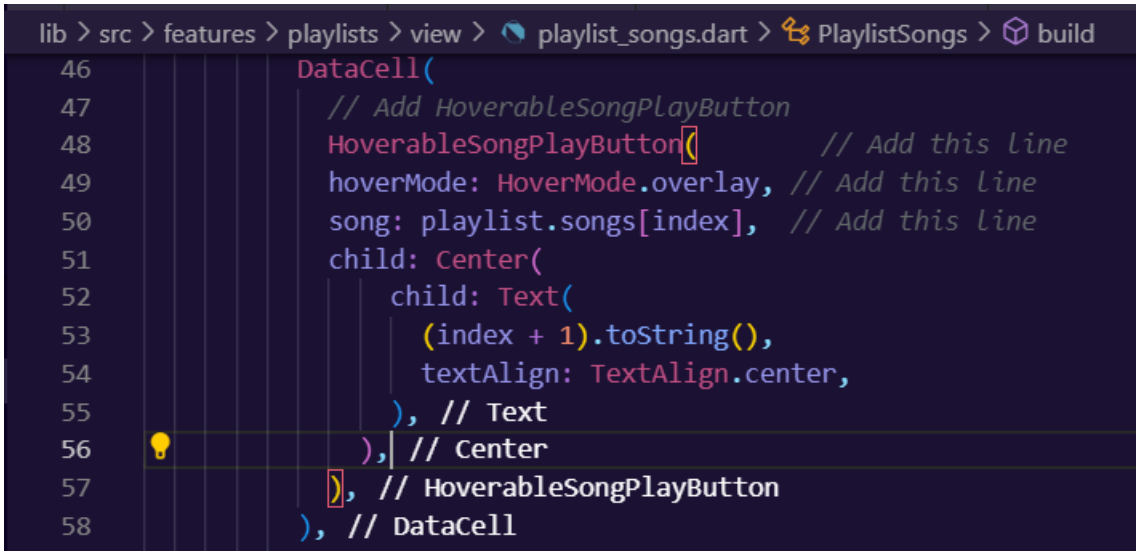
2. Hot reload the app and then hover over one of the recently played playlist tiles



The OutlinedCard changes opacity and rounds the corners.

3. Finally, animate the song number on a playlist into a play button using the HoverableSongPlayButton widget defined in

lib/src/shared/views/hoverable_song_play_button.dart. In lib/src/features/playlists/view/playlist_songs.dart, wrap the Center widget (which contains the song number) with a HoverableSongPlayButton:



Conclusion

We have learned that there are many small changes that we can integrate into an app to make it more beautiful, and also more accessible, more localizable, and more suitable for multiple platforms. These techniques include, but aren't limited to:

- Typography: Text is more than just a communication tool. Use the way that text is displayed to produce a positive effect on users' experience and perception of the app.
- Theming: Establish a design system that we can reliably use without having to make design decisions for every widget.
- Adaptivity: Consider the device and platform that the user is running the app on and its capabilities. Consider screen size and how the app is displayed.
- Motion and animation: Adding movement to the app adds energy to the user experience and, more practically, provides feedback for users.