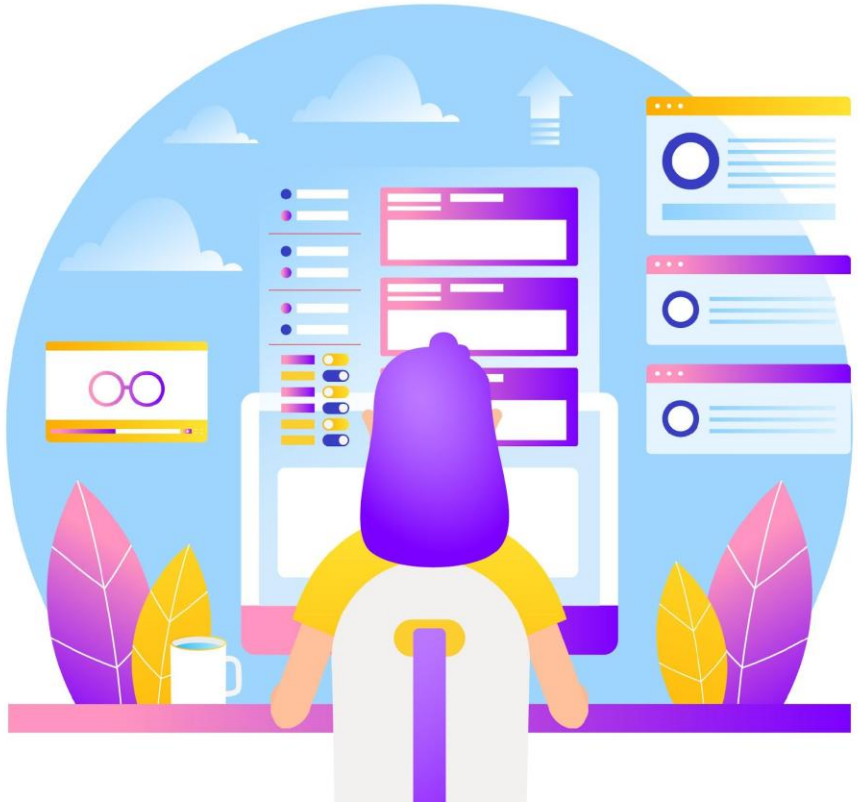
Advanced UI Development Techniques: Mastering the Art of Engaging and Scalable Interfaces

Unleashing the Power of Custom Widgets, Complex Layouts, Animations, State Management, and Backend Integration



Introduction

- Modern UI Challenge:
- Develop visually appealing, functional, scalable, and maintainable user interfaces across diverse platforms and devices.
- Advanced Techniques Requirement:
- Master advanced UI development techniques to overcome challenges and create exceptional user experiences.

Advanced Widget Usage

- Harnessing the Full Potential of Built-in Widgets
- Creating Custom Widgets for Specific Needs
- Leveraging Widget Libraries and Frameworks

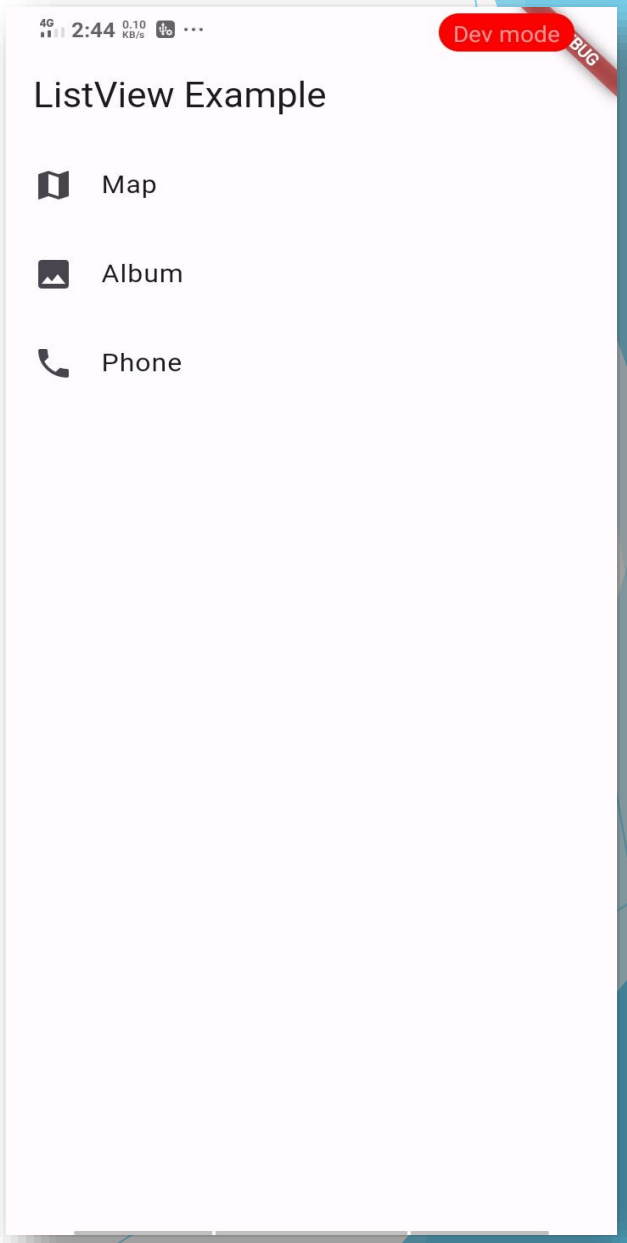
Built-in Widgets

- **ListView:**
- Displays items in a **one-dimensional layout:** either vertically (default) or horizontally.
- Useful for lists of text, images, cards, or any content that follows a linear order.
- Offers various constructors for different scenarios:
- ListView.builder for dynamically generating items from a data source.
- ListView.separated for adding separators between items.
- ListView.custom for full control over layout and scrolling.

Example of List view

```
15          body: ListView(
16            children: <Widget>[
17              ListTile(
18                leading: Icon(Icons.map),
19                title: Text('Map'),
20                onTap: () {
21                  // Add your onTap logic here
22                },
23              ), // ListTile
24              ListTile(
25                leading: Icon(Icons.photo),
26                title: Text('Album'),
27                onTap: () {
28                  // Add your onTap logic here
29                },
30              ), // ListTile
31              ListTile(
32                leading: Icon(Icons.phone),
33                title: Text('Phone'),
34                onTap: () {
35                  // Add your onTap logic here
36                },
37              ), // ListTile
38              // Add more ListTiles as needed
39            ], // <Widget>[]
40          ), // ListView
41        ), // Scaffold
```

Output:



- **Grid View:**

- **Grid View:**
- Arranges items in a **two-dimensional grid layout:** rows and columns.
- Suitable for photo galleries, product catalogs, or any visually oriented data.
- Includes constructors for defining the number of items per row/column and spacing between them.
- GridView.count specifies the number of items in each row or column.
- GridView.builder similar to List View. builder but arranges items in a grid.
- GridView.extent uses a fixed size for each item.

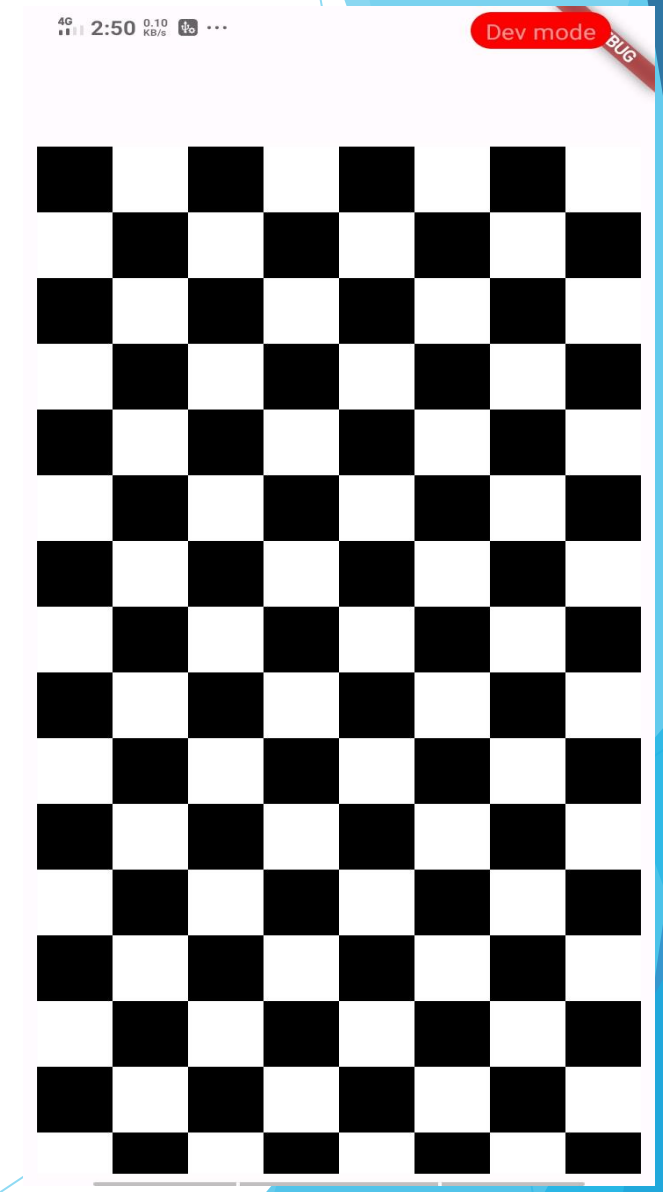
Example of Gid view

```dart
class gridLi extends StatefulWidget {
  const gridLi({Key? key}) : super(key: key);

  @override
  State<gridLi> createState() => _gridLiState();
}

class _gridLiState extends State<gridLi> {
  List<String>veggies = ['Broccoli','carrot','cucumber'];
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text(' '),
      ),
      body: Padding(
        padding: const EdgeInsets.all(8.0),
        child: GridView.builder(
          gridDelegate:
          SliverGridDelegateWithFixedCrossAxisCount(crossAxisCount: 8),
          itemBuilder: (context,index){
            var xIndex = index % 8;
            var yIndex = (index/8).floor();
            return Container(
              color: (xIndex + yIndex).isEven ? Colors.black : Colors.white,
              child: Stack(
                children: [],
              ),
            );
```

Output:



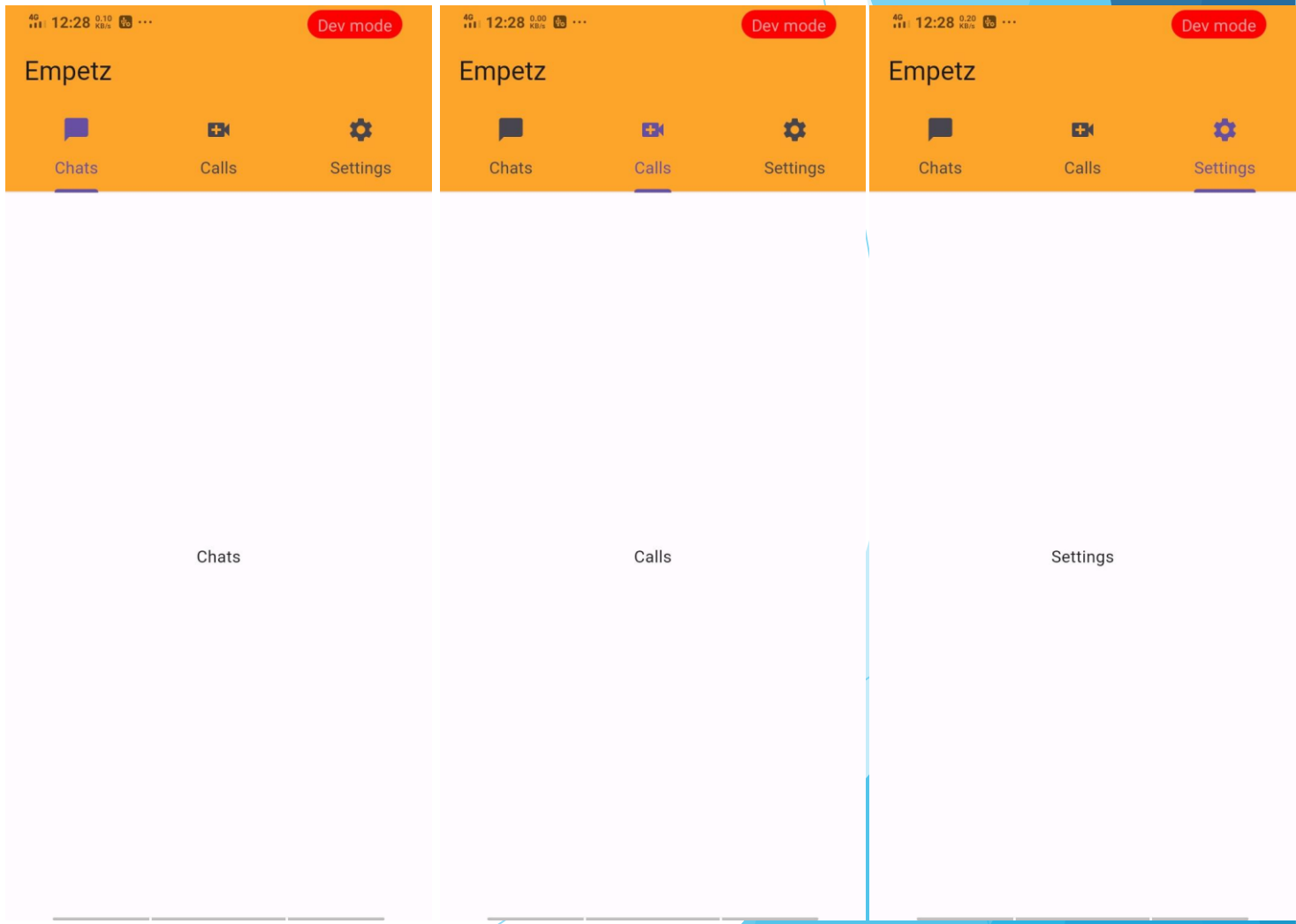
- **Tab Bar**

  - Working with tabs is a common pattern in apps that follow the Material Design guidelines.
  - Flutter includes a convenient way to create tab layouts as part of the material library.
  - This recipe creates a tabbed example using the following steps:
  - Create a Tab Controller.
  - Create the tabs.
  - Create content for each tab.

Example of Tab bar

```dart
class _TabbarExampleState extends State<TabbarExample> {
  @override
  Widget build(BuildContext context) {
    return DefaultTabController(
      length: 3,
      child: Scaffold(
        appBar: AppBar(
          title: const Text("Empetz"),
          backgroundColor: Color.fromARGB(255, 253, 165, 43),
          bottom: const TabBar(
            tabs: [
              Tab(
                icon: Icon(Icons.chat_bubble),
                text: "Chats",
              ), // Tab
              Tab(
                icon: Icon(Icons.video_call),
                text: "Calls",
              ), // Tab
              Tab(
                icon: Icon(Icons.settings),
                text: "Settings",
              ) // Tab
            ],
          ), // TabBar
        ), // AppBar
```

Output



Academy

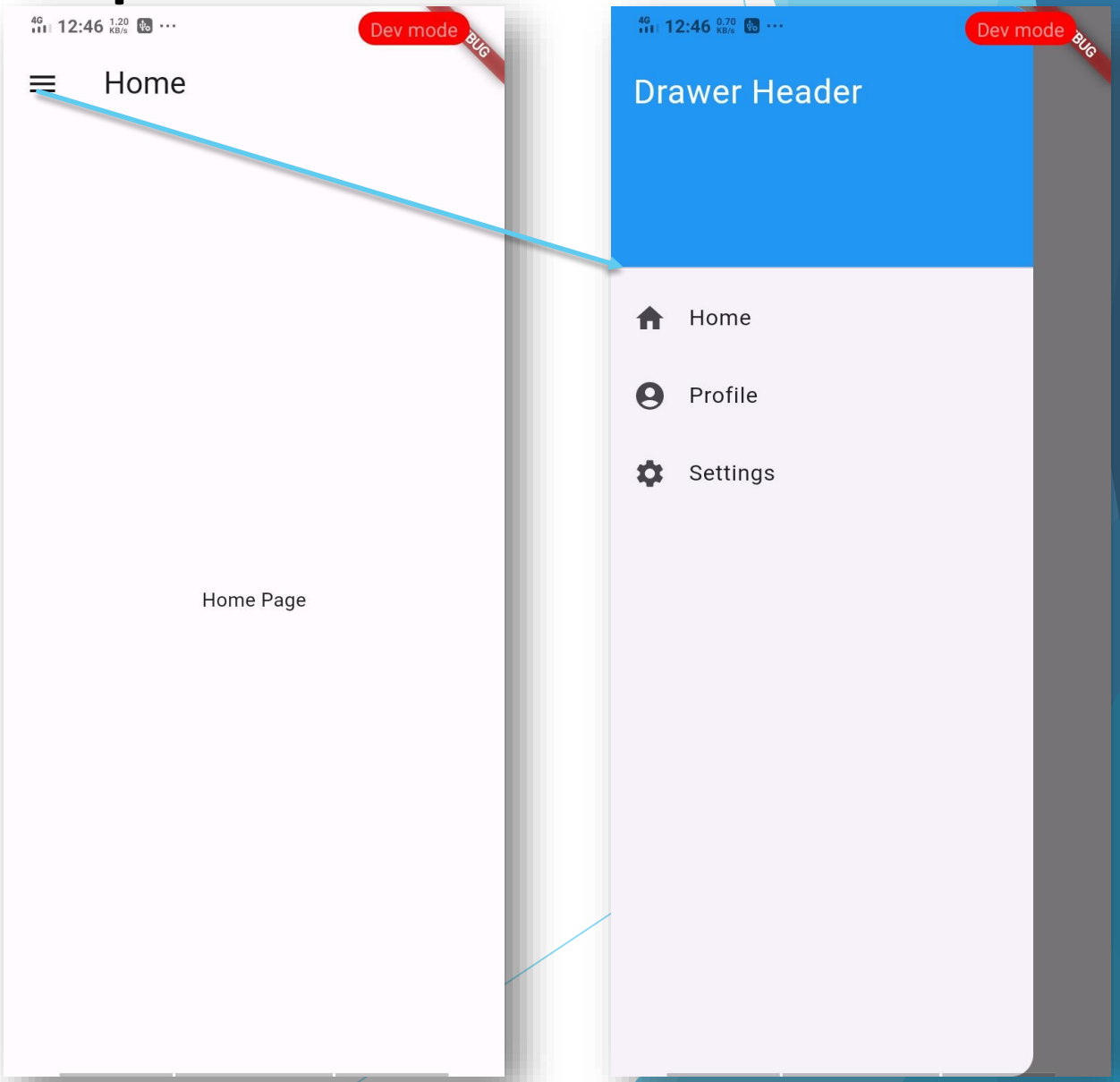
Navigation Drawer

- The navigation drawer in Flutter allows users to navigate to different pages of your app.
- The navigation drawer is added using the Drawer widget.
- It can be opened via swipe gesture or by clicking on the menu icon in the app bar.
- The navigation drawer can be used as an alternate option to the TabBar widget.
- It is recommended to use a navigation drawer when you have at least five pages to navigate.

Example of Navigation Drawer

```
29              DrawerHeader(
30                decoration: BoxDecoration(
31                  color: Colors.blue,
32                ), // BoxDecoration
33                child: Text(
34                  'Drawer Header',
35                  style: TextStyle(
36                    color: Colors.white,
37                    fontSize: 24,
38                  ), // TextStyle
39                ), // Text
40              ), // DrawerHeader
41              ListTile(
42                leading: Icon(Icons.home),
43                title: Text('Home'),
44                onTap: () {
45                  // Navigate to home screen
46                  Navigator.pop(context);
47                },
48              ), // ListTile
49              ListTile(
50                leading: Icon(Icons.account_circle),
51                title: Text('Profile'),
52                onTap: () {
53                  // Navigate to profile screen
54                  Navigator.pop(context);
55                  // Add your navigation logic here
56                },
57              ), // ListTile
```

Output:



Academy

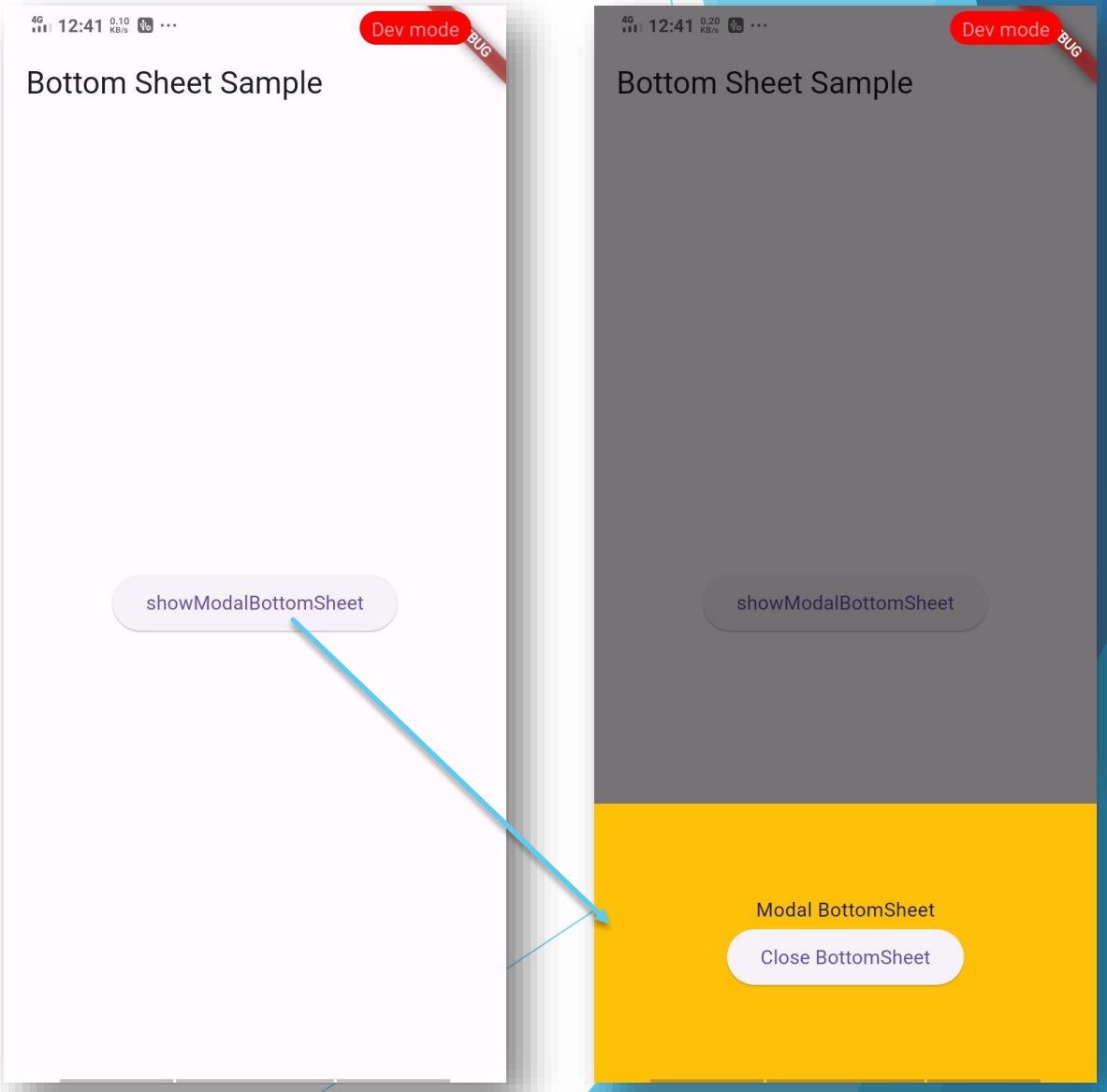
Bottom Sheet

- A bottom sheet is a UI component that slides up from the bottom of the screen to display additional content or options.
- It’s commonly used to present contextual actions, settings, or supplementary information without obstructing the main content of your app.
- Flutter’s Bottom sheet widget makes it easy to implement this interaction pattern.

Example of Bottom Sheet

```dart
@override
Widget build(BuildContext context) {
  return Center(
    child: ElevatedButton(
      child: const Text('showModalBottomSheet'),
      onPressed: () {
        showModalBottomSheet<void>(
          context: context,
          builder: (BuildContext context) {
            return Container(
              height: 200,
              color: Colors.amber,
              child: Center(
                child: Column(
                  mainAxisAlignment: MainAxisAlignment.center,
                  mainAxisSize: MainAxisSize.min,
                  children: <Widget>[
                    const Text('Modal BottomSheet'),
                    ElevatedButton(
                      child: const Text('Close BottomSheet'),
                      onPressed: () => Navigator.pop(context),
                    ), // ElevatedButton
                  ], // <Widget>[]
                ), // Column
              ), // Center
            ); // Container
```

Output:



Academy

Custom Widgets

- In Flutter, a custom widget is a user-defined component that encapsulates a specific UI element or functionality.
- They serve as the building blocks of your application, ensuring code reusability, maintainability, and a consistent design language.

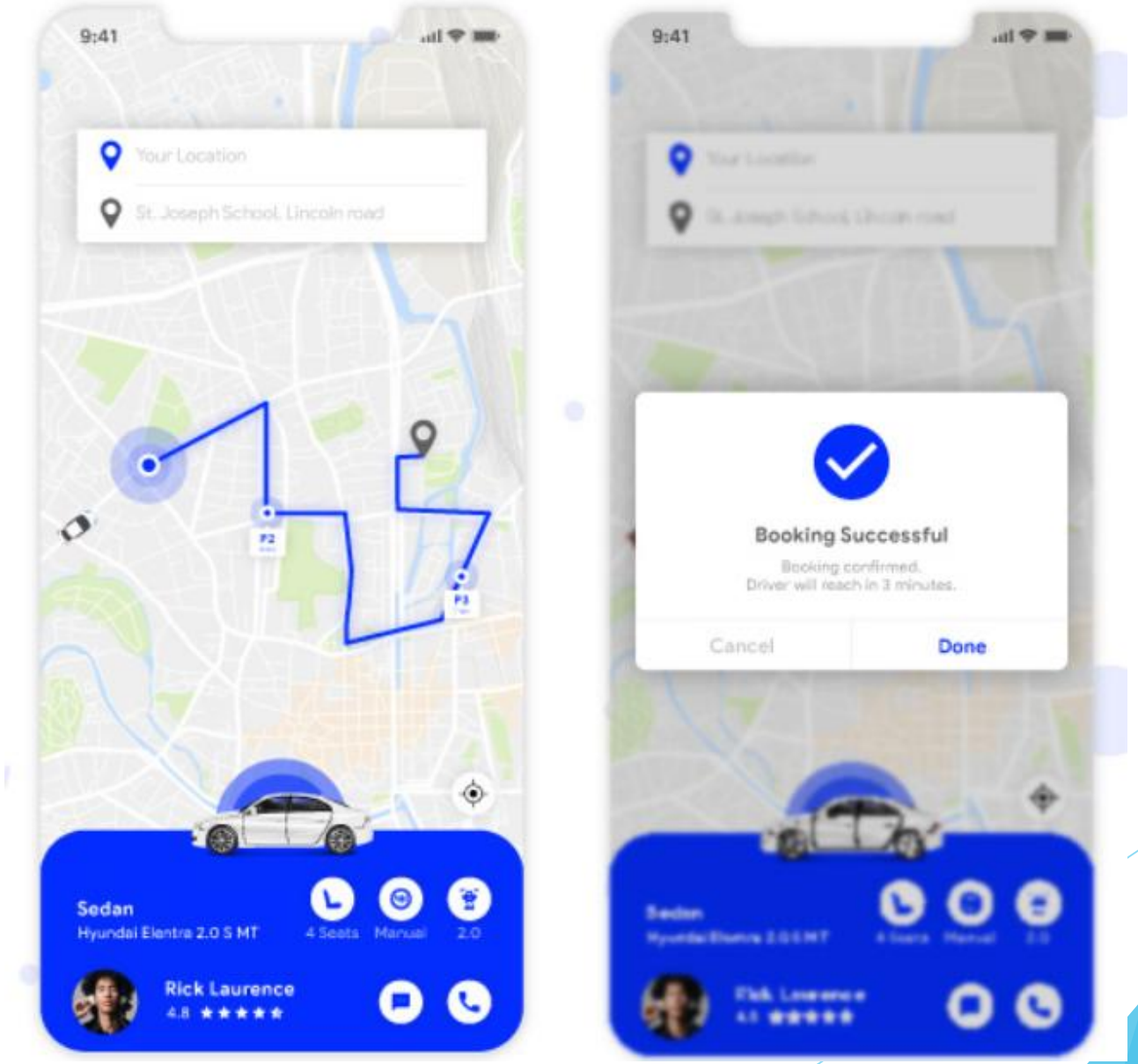
- **Essentially, you create a custom widget when:**
- You need a UI element that doesn't exist as a built-in widget in Flutter.
- You want to reuse the same UI element with different configurations or data throughout your app.
- You desire unique functionality or a customized look and feel for specific UI elements.

Complex Layouts and Responsive Design

- Designing Effective Layouts for Various Screen Sizes
- Mastering Layout Managers and Techniques
- Implementing Responsive Design Principles

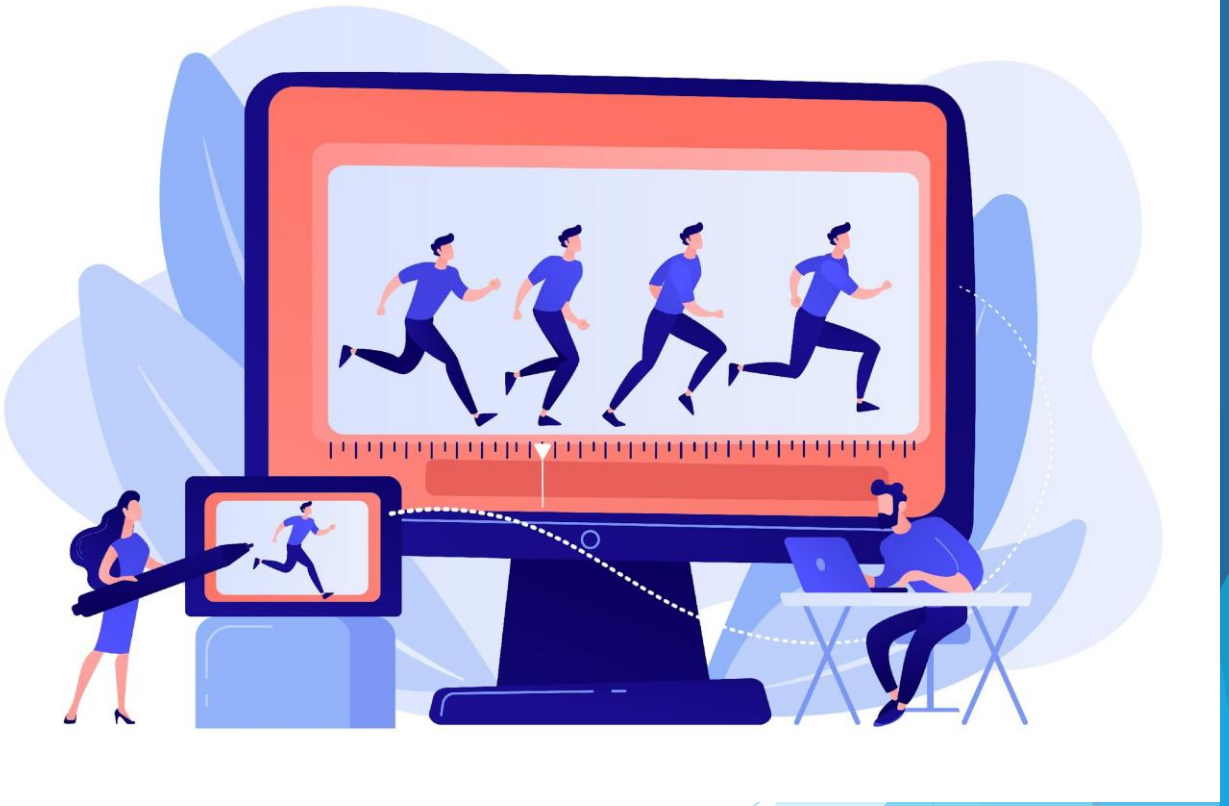
Aitrich Academy

Example: UI Design of an Taxi-Booking Application

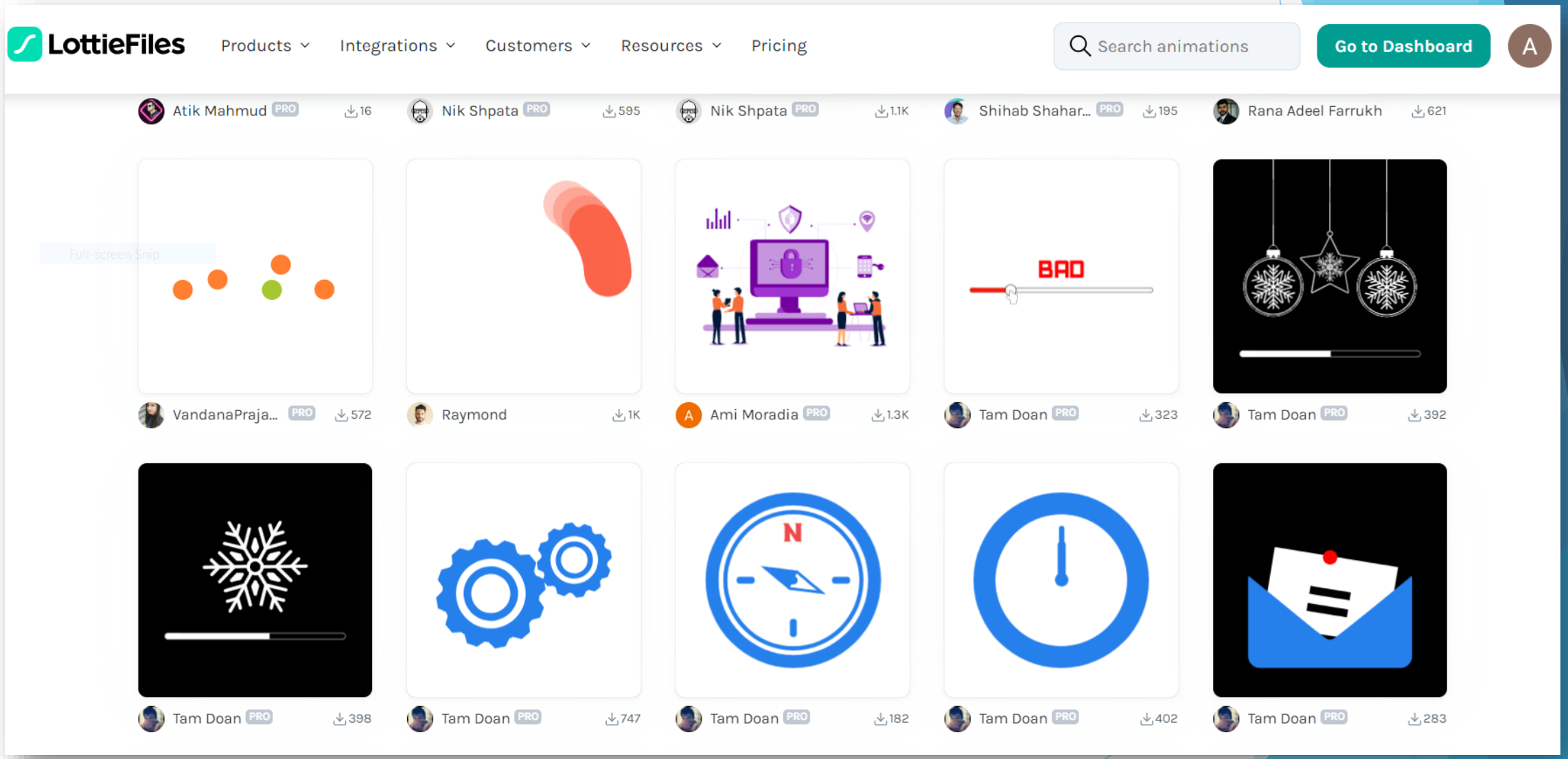


Advanced Animations and Motion Design

- Incorporating Advanced Animation Techniques
- Utilizing Motion Design Principles
- Employing Animation Libraries and Frameworks

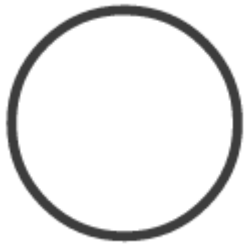


Example: Lottie file Animation



Aitrich Academy

Animations:



State Management at Scale

- Effectively Managing UI State in Large-Scale Applications
- Utilizing State Management Libraries and Frameworks
- Addressing State Complexity and Performance Challenges

Aitrich Academy

Integrating UI with Backend Services

- Connecting UI Components with Backend Services
- Implementing Data Fetching and Caching Strategies
- Handling Network Requests and Error Scenarios Gracefully

Conclusion

- Recap of Key Takeaways
- Emphasis on Mastering Advanced UI Development Techniques
- Encouragement for Continuous Learning and Experimentation