ĐẠI HỌC TÔN ĐỨC THẮNG
TON DUC THANG UNIVERSITY

502071

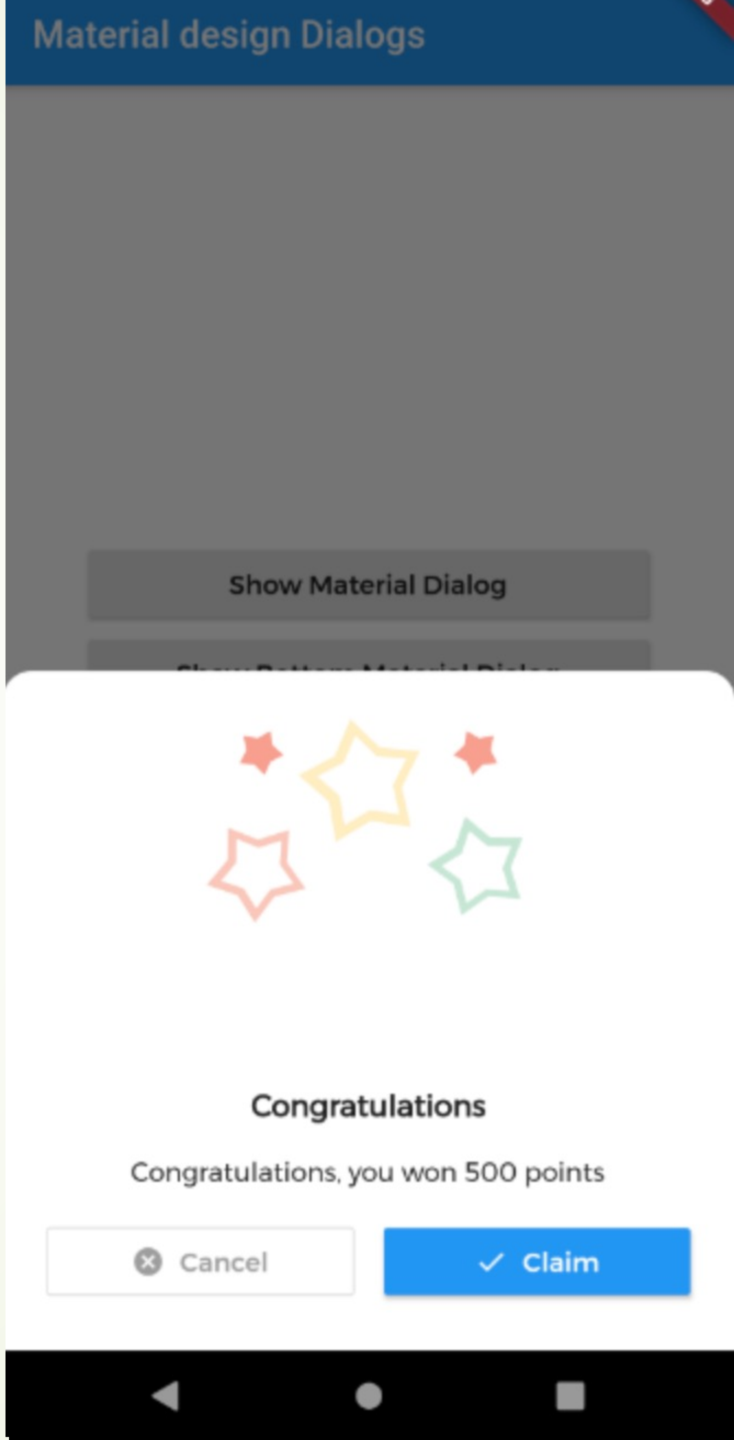
Cross-Platform Mobile Application Development

Dialogs

Flutter Dialog

Flutter Dialogs

- Dialogs are a way to present information or actions to the user in a separate window or overlay.
- Dialogs can be used to get user input, show progress, display information or errors, and confirm or cancel actions.
- Flutter provides several types of dialogs, they can be easily customized to fit the app's design and functionality.



Congratulations

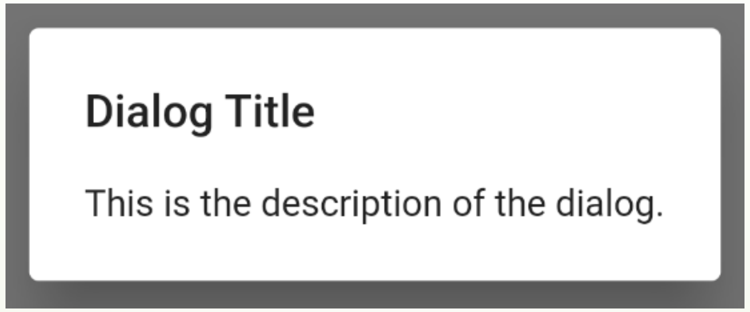
Congratulations, you won 500 points

Dialogs in Flutter

- Flutter provides several types of dialogs that can be used in an application. The most common types are:
  - Alert Dialog: This dialog presents a message to the user and requires them to take an action before dismissing it.
  - Simple Dialog: This dialog presents a list of options to the user and requires them to select one before dismissing it.
  - Modal Bottom Sheet: presents a list of options or content to the user and is displayed from the bottom of the screen.
  - Cupertino Dialog: This dialog presents a message to the user using the iOS-style design.
  - Cupertino Action Sheet: This dialog presents a list of options to the user using the iOS-style design.
- These dialogs can be customized with different styles, colors, and animations to match the app's design and brand.

Simple Dialog in Flutter

- A simple dialog is a type of dialog box that presents information to the user or prompts the user for a response. It is a lightweight dialog box that appears on top of the current screen and is used to get user input or display information without taking up too much space.
- A simple dialog typically contains a title, a message, and one or more buttons. The title is usually a short description of the purpose of the dialog box, and the message provides more detailed information. The buttons provide the user with options to respond to the dialog box.

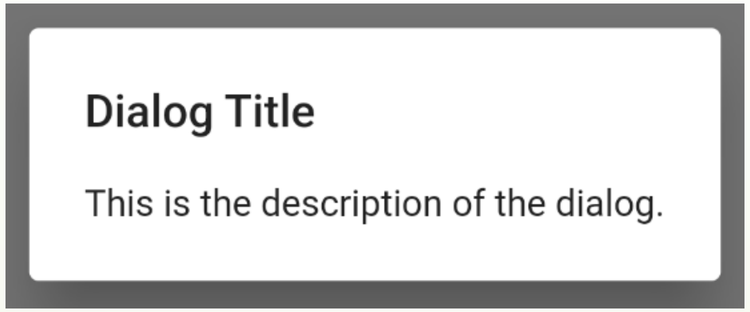


Dialog Title

This is the description of the dialog.

Simple Dialog in Flutter

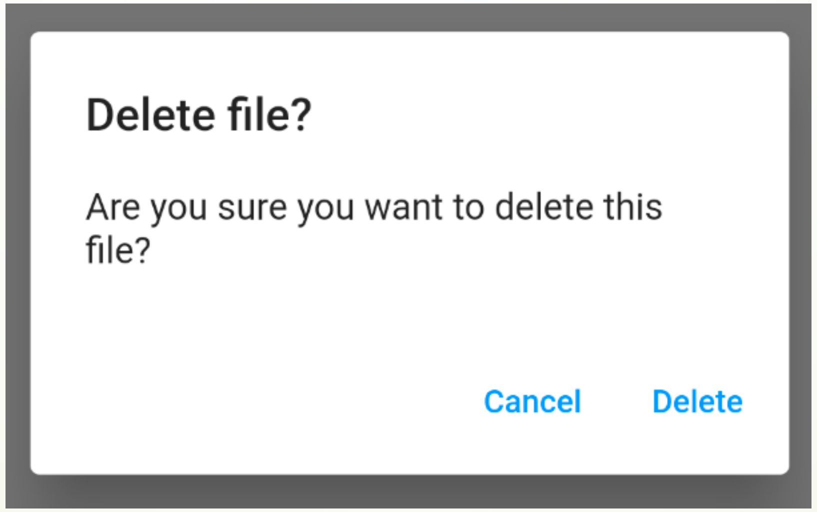
```
onPressed: () {
    showDialog(context: context, builder: (BuildContext context) {
            return AlertDialog(
                title: Text("Dialog Title"),
                content: Text("This is the description of the dialog."));
        },
    );
}
```


Dialog Title

This is the description of the dialog.

Alert Dialog in Flutter

- an Alert Dialog is a type of popup that displays important information or prompts the user to take a specific action. It typically contains a message and one or more buttons that allow the user to respond to the message or complete the action.
- Alert Dialogs are commonly used to confirm an action, such as deleting a file or closing an application. They can also be used to display important information, such as an error message or a warning.

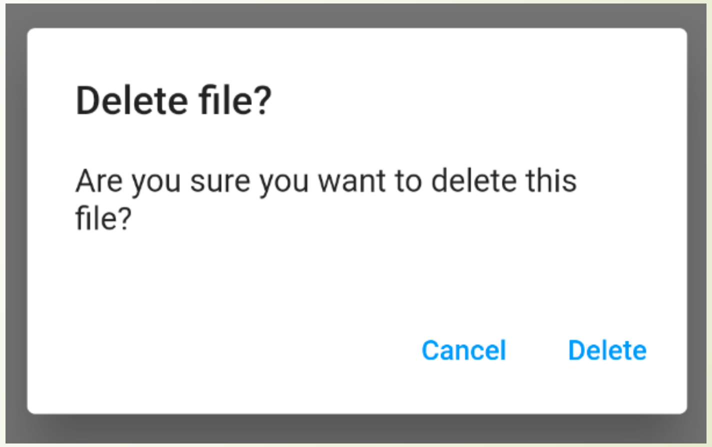


Delete file?

Are you sure you want to delete this file?

Alert Dialog in Flutter

```
showDialog(context: context, builder: (BuildContext context) {
    return AlertDialog(
        title: Text("Delete file?"),
        content: Text("Are you sure you want to delete this file?"),
        actions: [
            TextButton(child: Text(”Cancel"), onPressed: () {
                Navigator.of(context).pop();
            }),
            TextButton(child: Text(”Delete"), onPressed: () {
                Navigator.of(context).pop();
            }),
        ],
    );
},
);
```


Delete file?

Are you sure you want to delete this file?

showDialog function

- The showDialog() function takes several parameters that you can use to customize the appearance and behavior of the dialog.
  - bool barrierDismissible: determines whether the user can dismiss the dialog box by tapping outside of it.
  - barrierColor: A color value that determines the color of the barrier that is displayed behind the dialog. The default value is Colors.black54, which provides a semi-transparent black color. You can set this parameter to any color value you like.
  - bool useSafeArea: determines whether the dialog should be displayed within a SafeArea widget.

Custom Dialog

- You can create a custom dialog by using the showDialog() function and passing a custom widget as the builder parameter. Here are the steps to create a custom dialog.
    1. Create a new widget class that extends the StatelessWidget or StatefulWidget class, this widget will contain the content of your custom dialog.

```
class MyCustomDialog extends StatelessWidget {

    @override
    Widget build(BuildContext context) {
        return Container(
            // Add your dialog content here
        );
    }
}
```

Custom Dialog

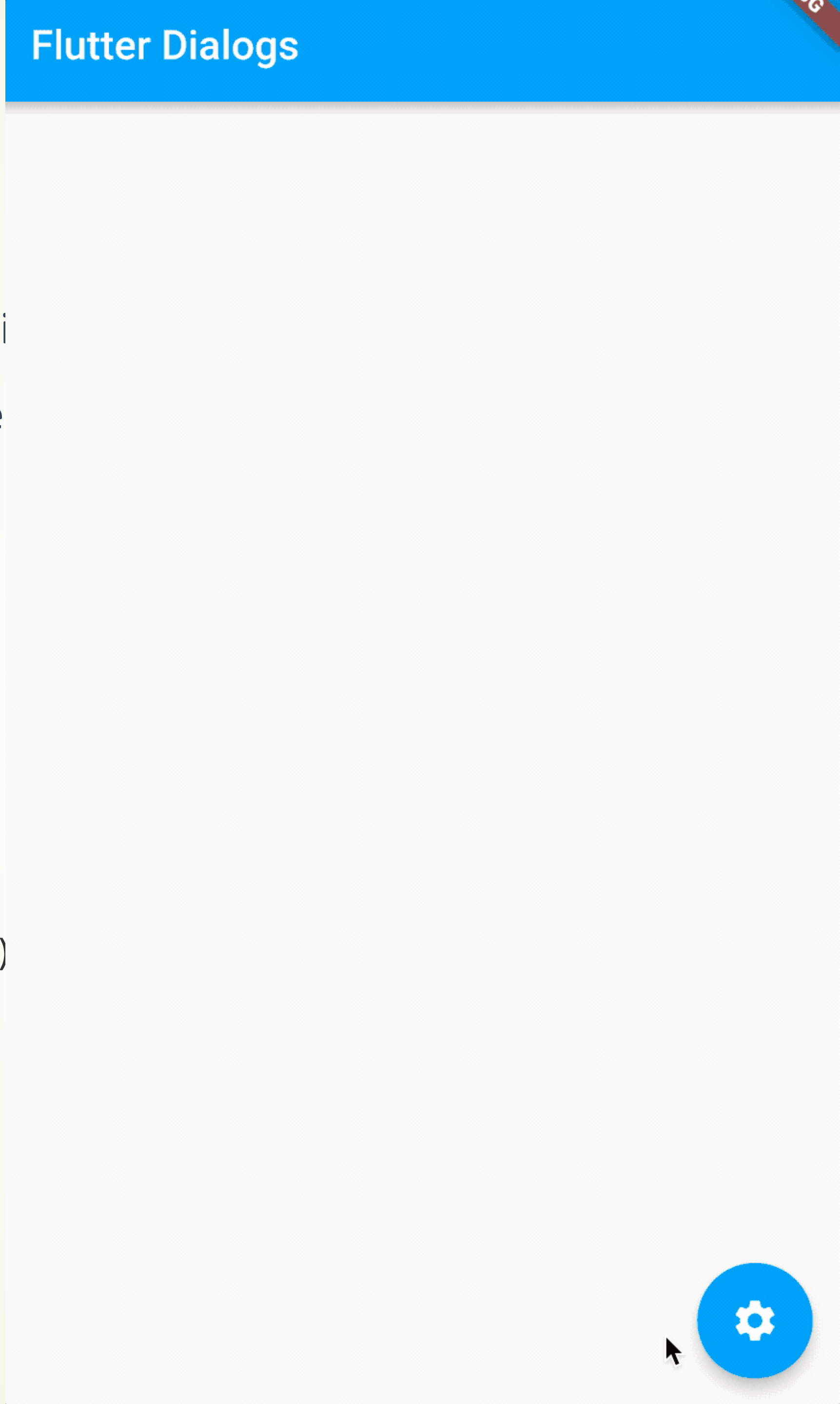
2. In your main widget or screen, call the showDialog() function and pass an instance of your custom widget as the builder parameter.

```
showDialog(
    context: context,
    builder: (BuildContext context) {
        return MyCustomDialog();
    },
);
```

Custom Dialog

3. Inside your custom widget, add the content that you want to di
include text, images, buttons, or any other widgets that you ne

```
class MyCustomDialog extends StatelessWidget {
    Widget build(BuildContext context) {
        return Dialog(child: Column(
                mainAxisSize: MainAxisSize.min,
                children: [
                    Text('My Image',),
                    Image.network('https://image.jpg')
                ],
            ),
        );
    }
}
```



Dialog vs AlertDialog widgets

- In Flutter, both Dialog and AlertDialog are classes that allow you to display a dialog box on top of your app's content. Here's a brief explanation of the differences between the two:
  - Dialog: The Dialog class is a generic dialog box that can be used to display any type of content. It is a more flexible option than AlertDialog because it allows you to customize the appearance and behavior of the dialog to fit your specific use case.
  - AlertDialog: The AlertDialog class is a pre-built dialog box that is designed specifically for displaying alerts or notifications to the user. It has a standard design that follows the Material Design guidelines and includes a title, message, and one or more action buttons.

showDialog() result

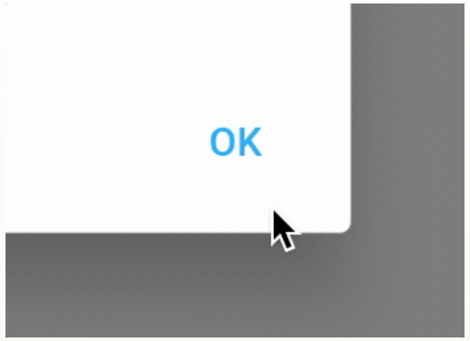
- It's a common practice to use showDialog to prompt the user for input, confirmation, or to display important information.
- When using showDialog, you may want to get a result back from the user, such as a button press or text input. Here's how to do that:
  - Call the showDialog() as an async function with the await keyword and assign the return value to a variable.

```
void _showSingleChoiceDialog() async {
    var data = await showDialog(...)
    // do something with return 'data'
}
```

showDialog() result

- Inside the dialog action buttons onPress or onTap event, call the Navigator pop() method to close the dialog and pass any value to the pop() method as your return value.

```
TextButton(
    child: Text('OK'),
    onPressed: () {
        Navigator.of(context).pop('Returned value');
    },
),
```



Login Dialog

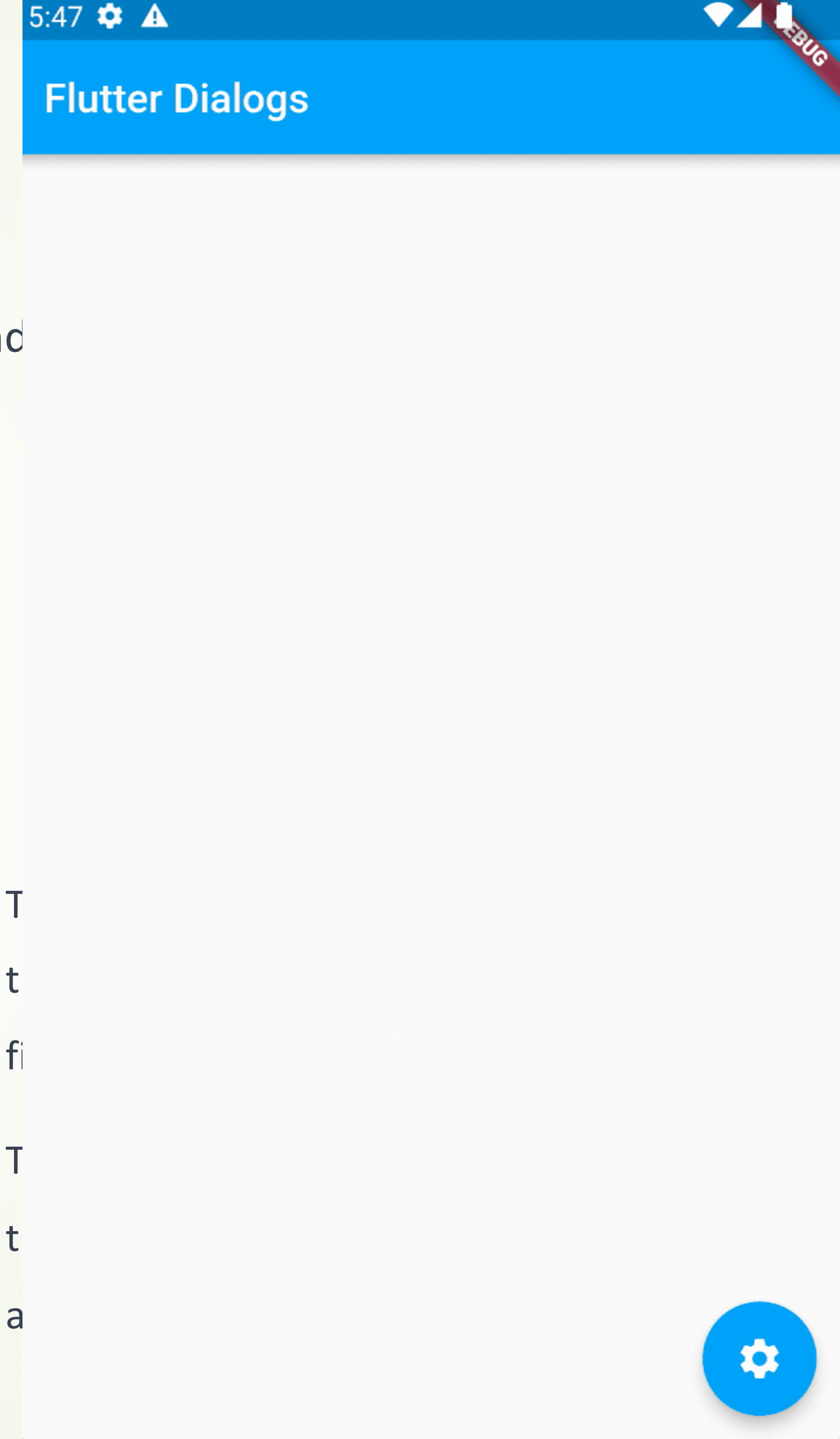
Create a Login Dialog

- You can create a login dialog using the showDialog() function and

  the login form.

```
class LoginDialog extends StatefulWidget {
    _LoginDialogState createState() => _LoginDialogState();
}

class _LoginDialogState extends State<LoginDialog> {
    Widget build(BuildContext context) {
        return AlertDialog(
            title: Text('Login'),
            content: Form(...),
            actions: [...],
        );
    }
}
```

- T
  t
  fi
- T
  t
  a



Create a Login Dialog

- The content of the AlertDialog is a Form widget that contains two TextFormField widgets for email and password. The Form widget handles validation and saving of form data.

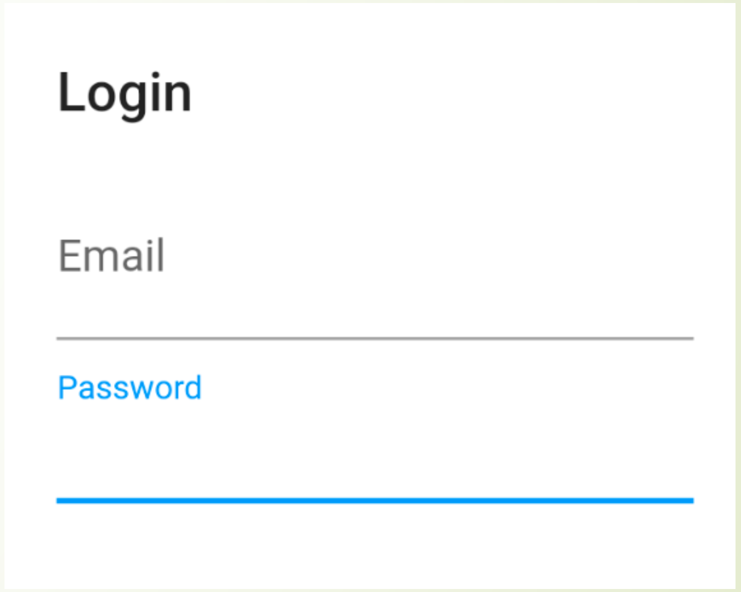
```
class _LoginDialogState extends State<LoginDialog> {
    final _formKey = GlobalKey<FormState>();
    String? _email = '';
    String? _password = '';

    Widget build(BuildContext context) {
        return AlertDialog(
            title: Text('Login'),
            content: Form(
                key: _formKey,
                child: Column(...)
            ),
            actions: [...],
        );
    }
}
```

- The _formKey is a GlobalKey<FormState> that uniquely identifies the Form widget. We use it to access the state of the form and perform form validation and submission.

Create a Login Dialog

- The content of the AlertDialog is a Form widget that contains two TextFormField widgets for email and password. The Form widget handles validation and saving of form data.

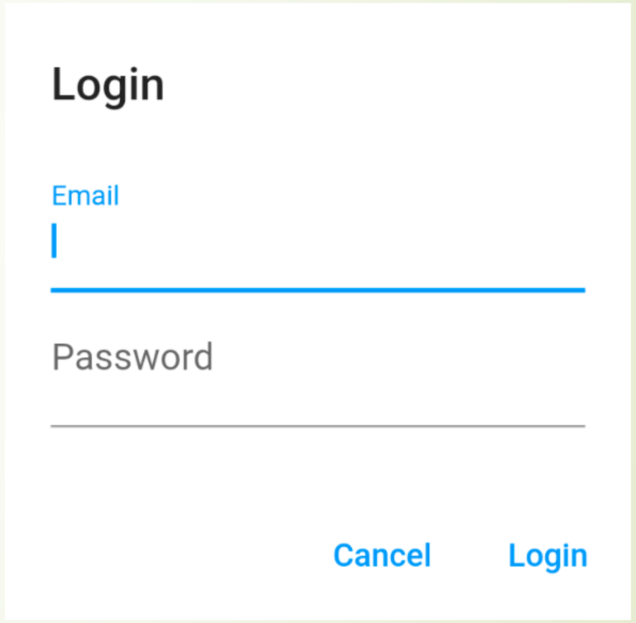
```
Column(children: [
        TextFormField(
            validator: (value) {
                if (value?.isEmpty ?? false) {
                    return 'Please enter your email.';
                }
                return null;
            },
            onSaved: (value) { _email = value;},
        ),
        TextFormField(
            validator: (value) {
                if (value?.isEmpty ?? false) {
                    return 'Please enter your password.';
                }
                return null;
            },
            onSaved: (value) { _password = value;},
        ),
    ],
```



Create a Login Dialog

- When the user clicks the "Login" button, we validate the form by calling _formKey.currentState.validate(). If the form is valid, we save the form data by calling _formKey.currentState.save()

```
actions: [
    TextButton(child: Text('Cancel’),
        onPressed: () {
            Navigator.of(context).pop();
        },
    ),
    TextButton(child: Text('Login’),
        onPressed: () {
            if (_formKey.currentState?.validate() ?? false) {
                _formKey.currentState?.save();
                // Call login API here
                Navigator.of(context).pop();
            }
        },
    ),
```



Create a Login Dialog

- You can create a login dialog using the showDialog() function and a custom widget that contains the login form.

```
class LoginDialog extends StatefulWidget {
    _LoginDialogState createState() => _LoginDialogState();
}

class _LoginDialogState extends State<LoginDialog> {
    Widget build(BuildContext context) {
        return AlertDialog(
            title: Text('Login'),
            content: Form(...),
            actions: [...],
        );
    }
}
```

```
showDialog(
    context: context,
    builder: (BuildContext context) {
        return LoginDialog();
    },
);
```

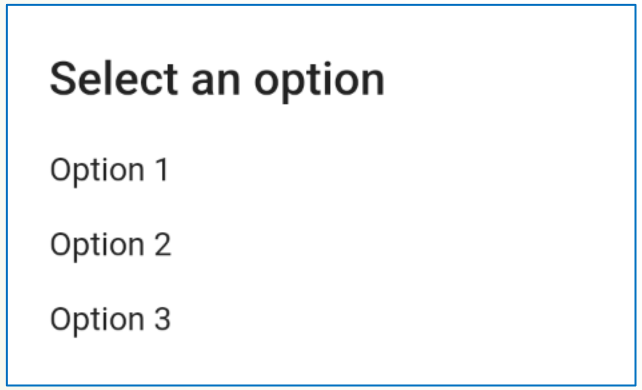
Create a Login Dialog

- Overall, the login dialog is a useful way to prompt the user for their login credentials in a secure and user-friendly manner. By using the AlertDialog and Form widgets, we can create a custom dialog that handles form validation and submission with minimal code.

Single Choice Dialog

Single Choice Dialog

- A single choice dialog is a type of dialog box or pop-up window that presents the user with a list of options from which they can select one. The user can choose only one option from the list, hence the name "single choice".

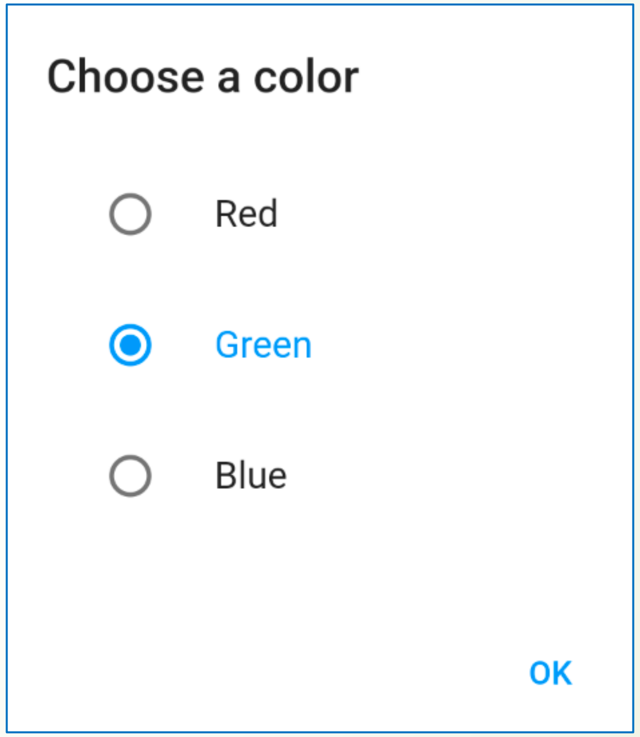


Select an option

Option 1

Option 2

Option 3



Choose a color

○ Red

● Green

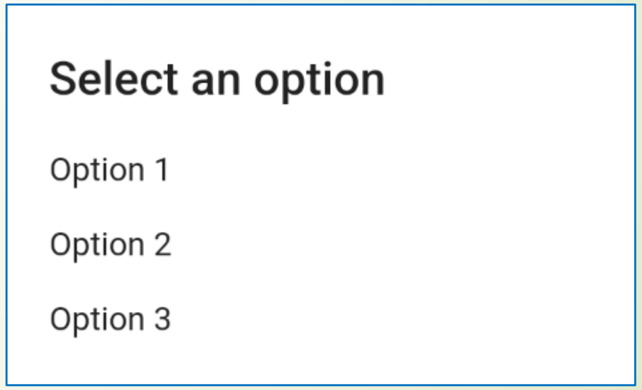
○ Blue

OK

Single Choice Dialog

- In this example, we displays a SimpleDialog with three options. When an option is selected, the dialog is closed and the selected option is returned using the Navigator.pop() method.

```
SimpleDialog(
     title: Text('Select an option'),
     children: [
          SimpleDialogOption(
               onPressed: () { Navigator.pop(context, 'Option 1’); },
               child: Text('Option 1')),
          SimpleDialogOption(
               onPressed: () { Navigator.pop(context, 'Option 2');},
               child: Text('Option 2'))
     ],
)
```


Select an option

Option 1

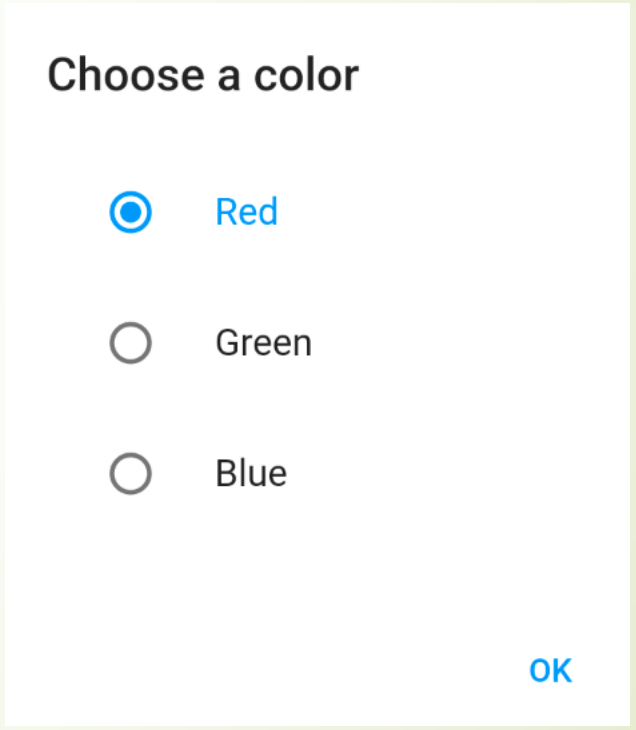
Option 2

Option 3

Single Choice Dialog

- You can also use the AlertDialog widget and ListTiles widget to create this dialog.

```
AlertDialog(
    title: Text('Choose a color'),
    content: Column(mainAxisSize: MainAxisSize.min,
    children: [
        RadioListTile(
            value: ‘Red',
            title: Text(‘Red'),
            selected: true,
            onChanged: (v) => {}, )
        ],
    actions: [
        TextButton( child: Text('OK'),
        onPressed: () {},
    ],
)
```


Choose a color

Red

Green

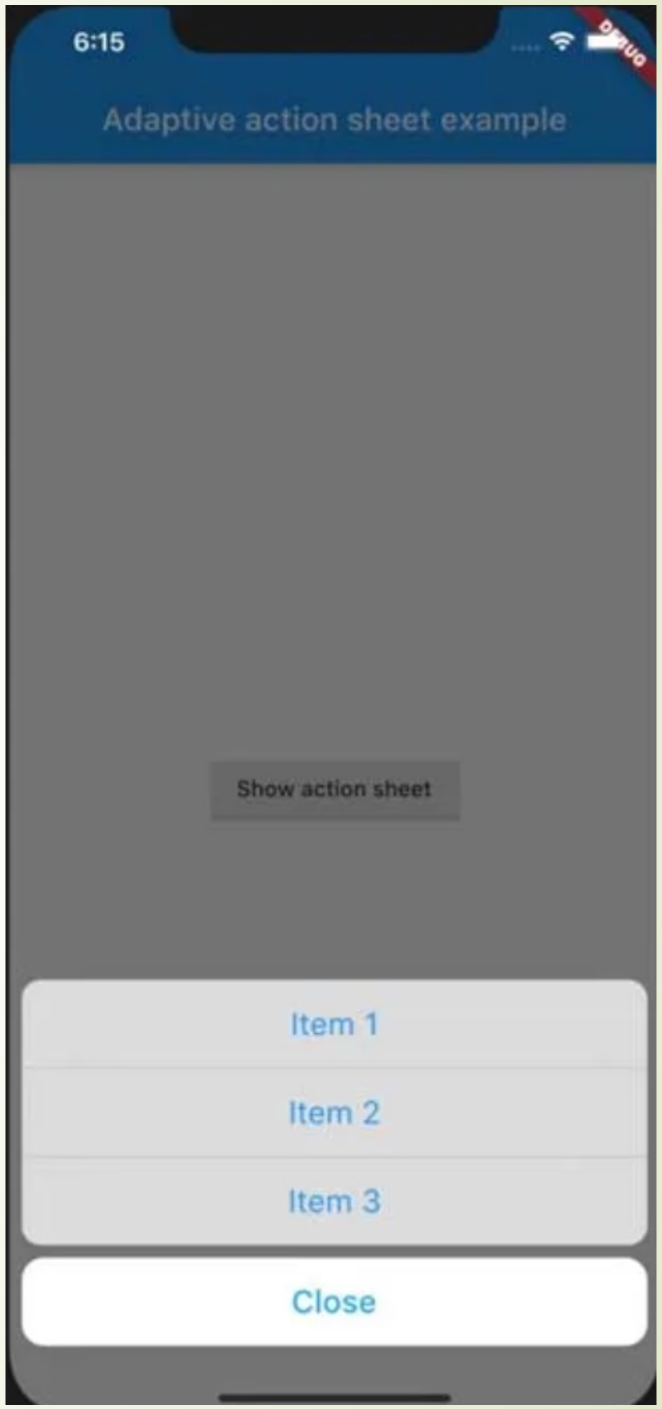
Blue

OK

Bottom Sheet

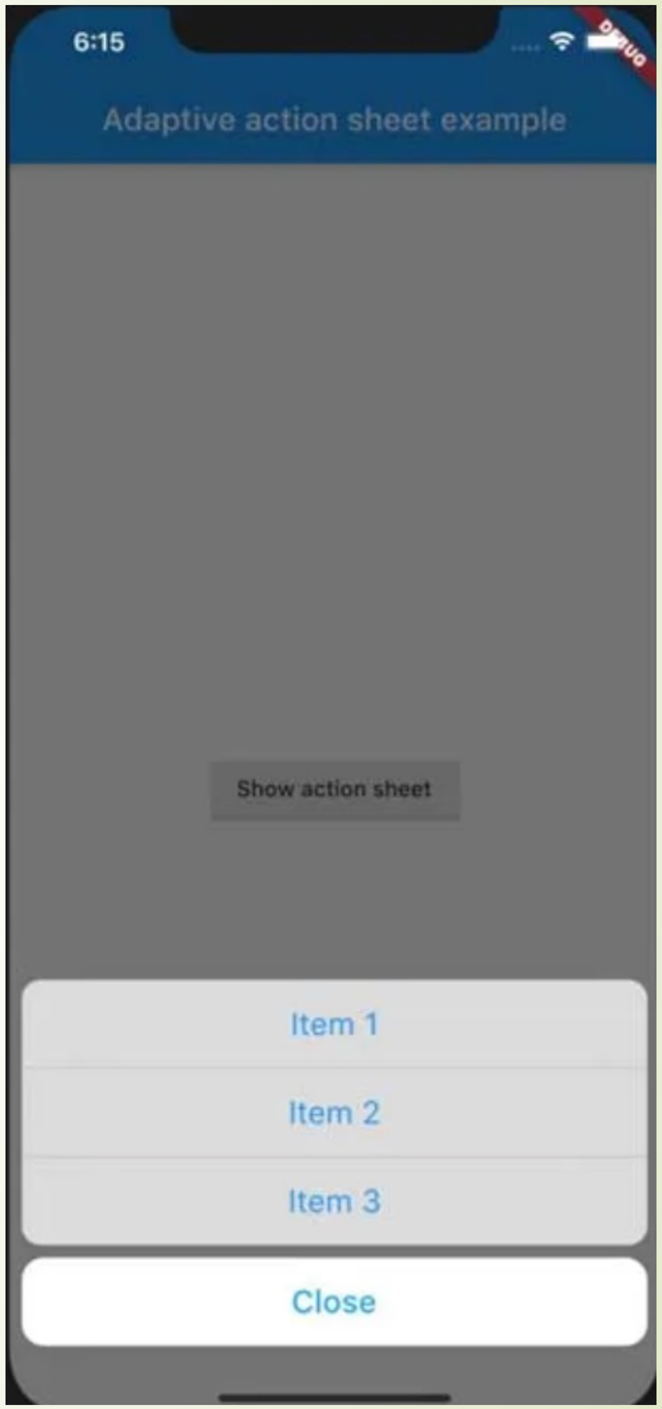
Bottom Sheet

- A bottom sheet is a type of modal sheet that slides up from the bottom of the screen to reveal additional content or options. It can be used to display menus, settings, or any other type of content that is not the primary focus of the screen.



Bottom Sheet

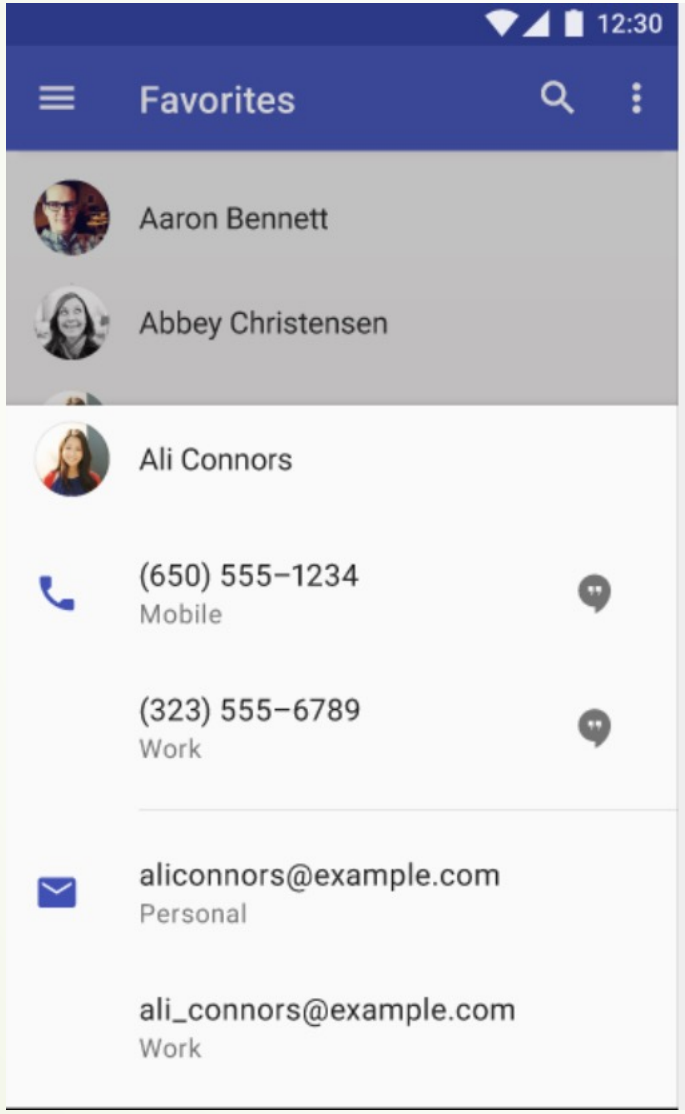
- Modal bottom sheets are very common in mobile apps. They are often used to display links to other apps on the user's mobile device.
- A modal bottom sheet will appear in response to some user action, such as tapping an icon. It can be dismissed by any of the following user actions:
  - Tapping an item within the bottom sheet
  - Tapping the main UI of the app
  - Swiping down

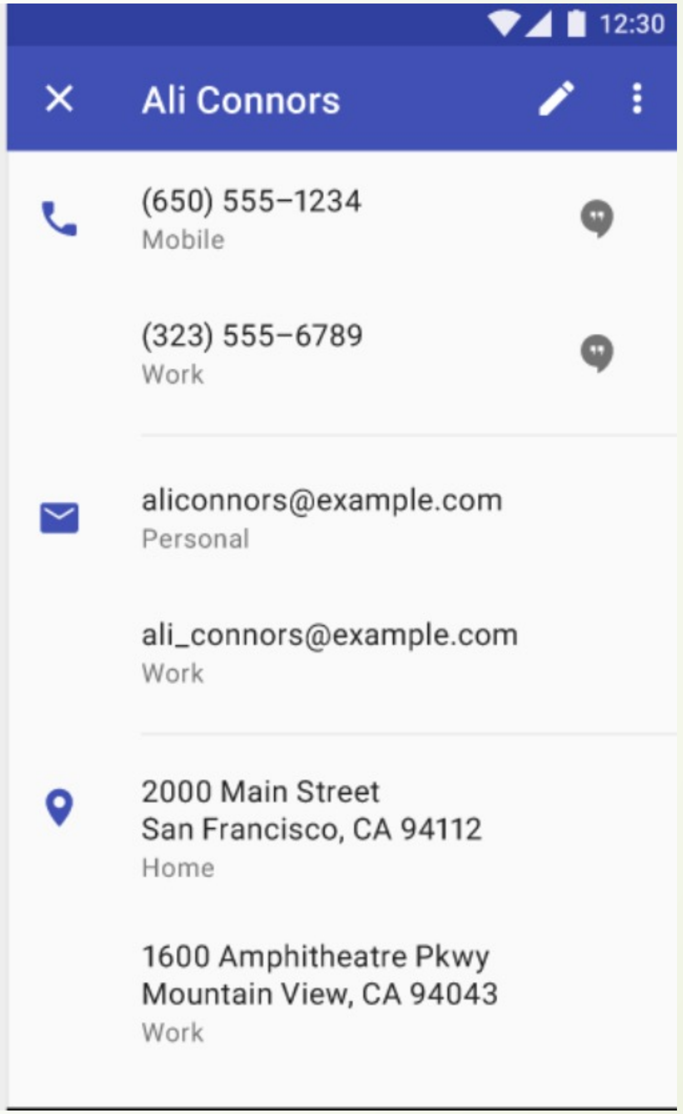


Bottom Sheet

- There are two types of bottom sheets available in Flutter:
  - Persistent Bottom Sheet: A persistent bottom sheet remains visible even when the user interacts with the rest of the app. It can be used to display content that is relevant throughout the app's lifecycle, such as a music player or a search bar.
  - Modal Bottom Sheet: A modal bottom sheet is displayed as a modal dialog and takes up the full screen height. It is typically used to display context-specific information or actions, such as a confirmation dialog or a menu.

Bottom Sheet





Modal Bottom Sheet

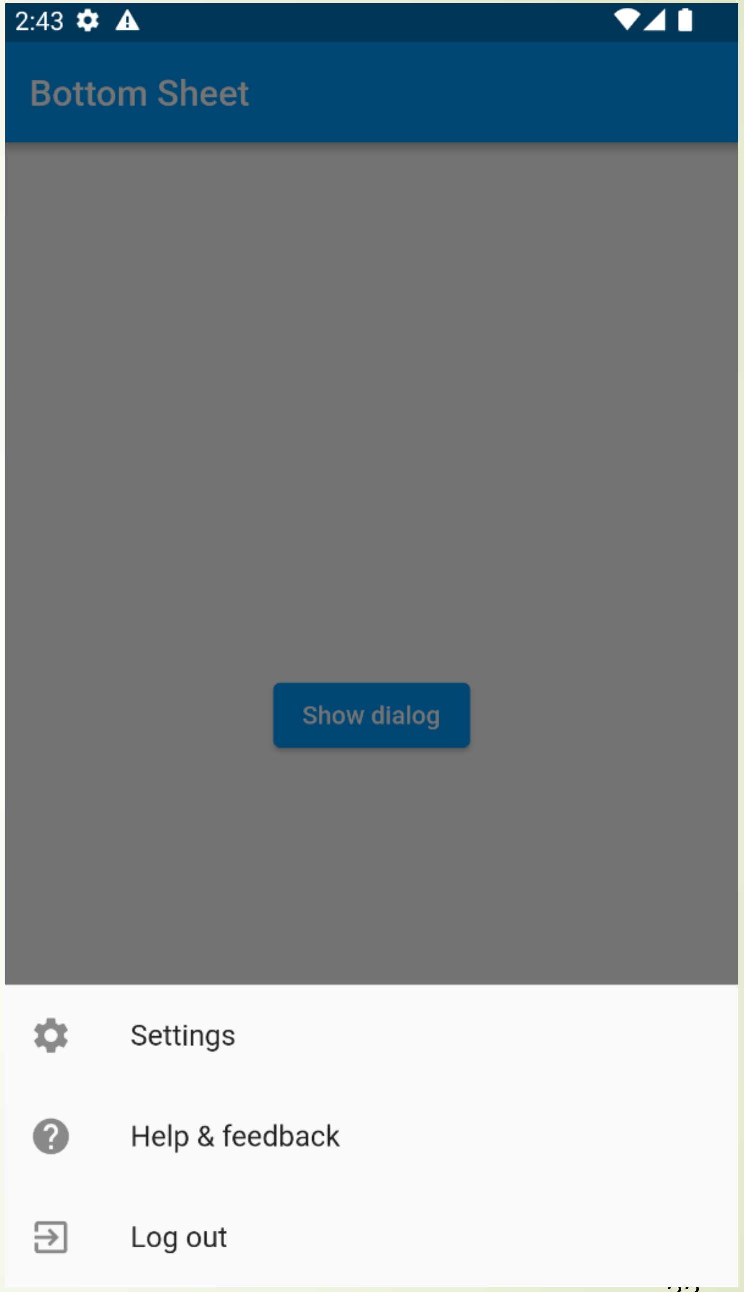
- A bottom sheet can be displayed using the showModalBottomSheet method, which takes a BuildContext and a builder function as arguments. The builder function returns the content of the bottom sheet, which can be any widget or combination of widgets.

```
showModalBottomSheet(
    context: context,
    builder: (BuildContext context) {
        return Container(
            child: Text('This is a bottom sheet'),
        );
    },
);
```

Bottom Sheet

- Here's an example of creating a simple bottom sheet in Flutter that contains a list of ListTile items:

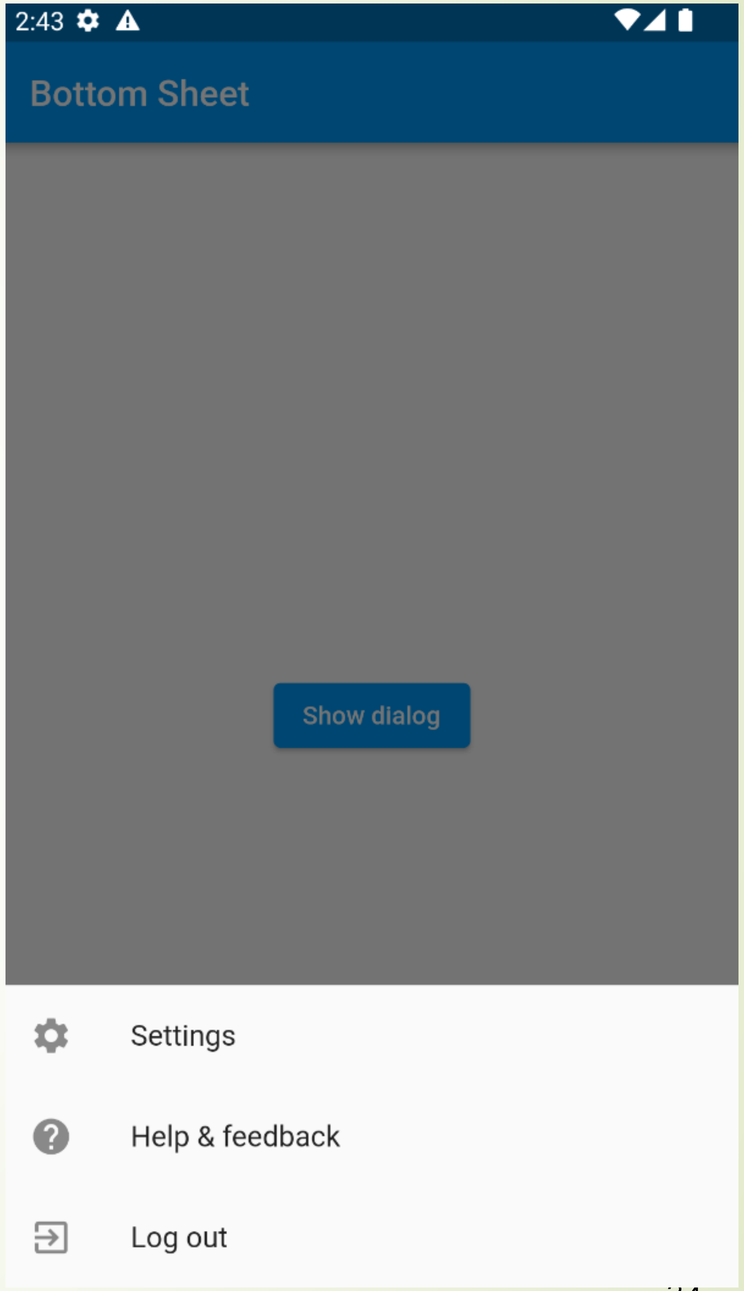
```
showModalBottomSheet(
    context: context,
    builder: (BuildContext context) {
        return Wrap(
            children: [
                ListTile(
                    leading: Icon(Icons.exit_to_app),
                    title: Text('Log out'),
                    onTap: () {},
                ),
            ],
        );
    },
);
```



Bottom Sheet

- When the use tap on each of the items, you often want to close the dialog by calling Navigator.pop(context) method.
- You can optionally pass a return value to the caller of bottom sheet by passing the return value as the second argument. Make sure to show bottom sheet with the await keyword.

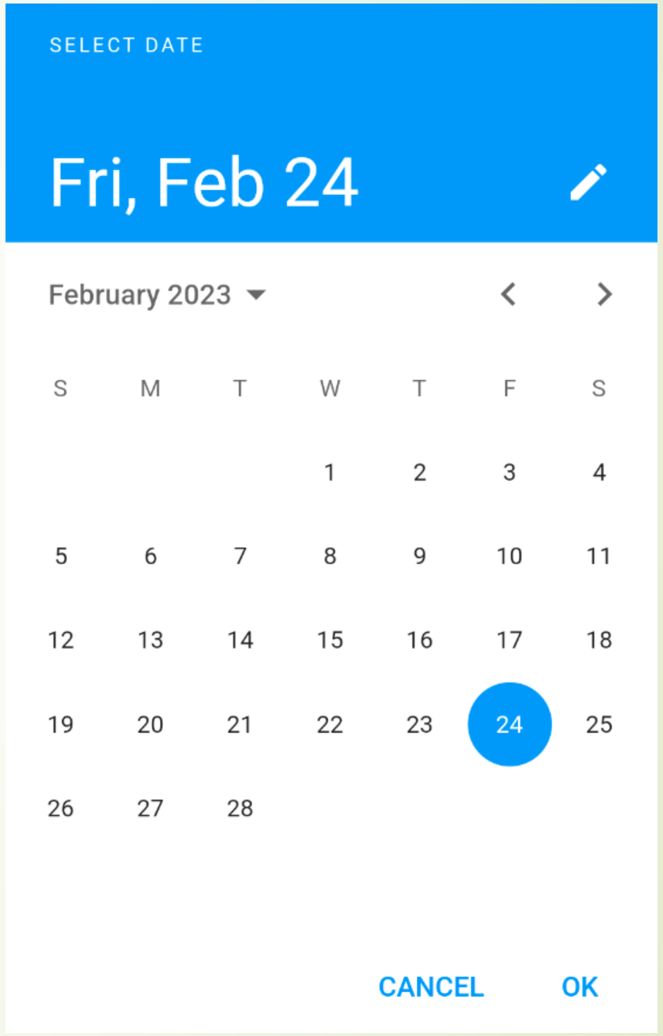
```
void _showBottomSheet(BuildContext context) async {
    String option = await showModalBottomSheet(…)
}

ListTile(
    leading: Icon(Icons.help),
    title: Text('Help & feedback'),
    onTap: () {
        Navigator.pop(context, 'Help & feedback');
    },
),
```



Picker Dialogs

Flutter Picker Dialogs

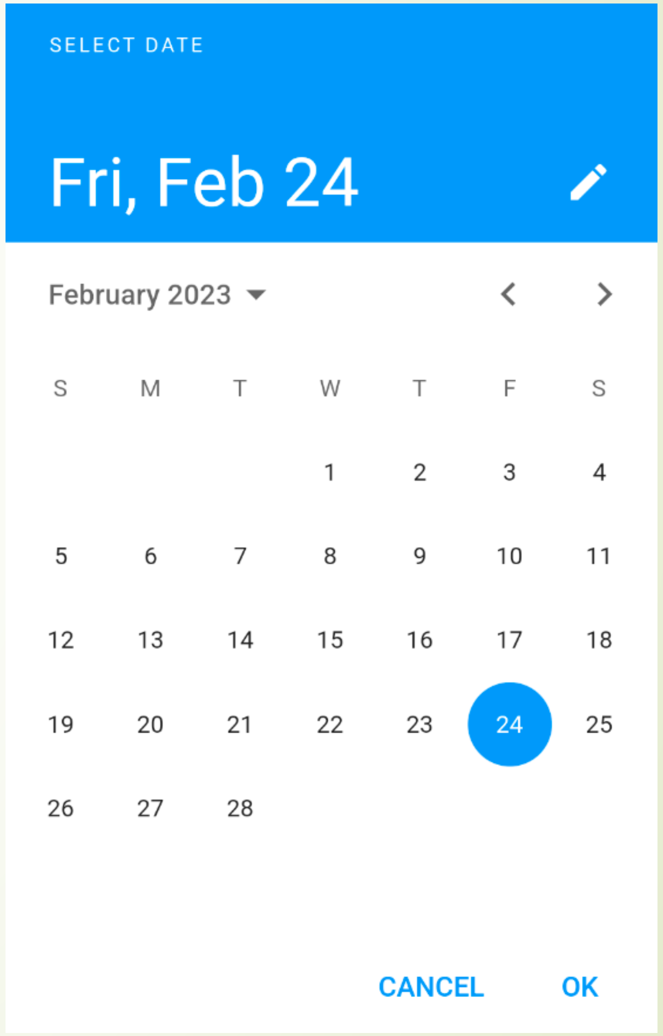
- Pickers are a common UI component in mobile apps that allow the user to select a value from a set of options. In Flutter, there are several types of pickers available for developers to use, each with its own set of features and design patterns.
- Date and Time Pickers are a common UI component in mobile apps that allow the user to select a specific date or time. In Flutter, there are two built-in picker widgets that can be used for this purpose: showDatePicker and showTimePicker.



Date Picker

- The showDatePicker method displays a dialog that allows the user to select a date. The dialog provides controls for changing the year, month, and day, and can be customized with options such as the initial date, minimum and maximum dates, and the text that appears on the confirmation and cancel buttons.

```
showDatePicker(
    context: context,
    initialDate: DateTime.now(),
    firstDate: DateTime(2015, 8),
    lastDate: DateTime(2101),
);
```



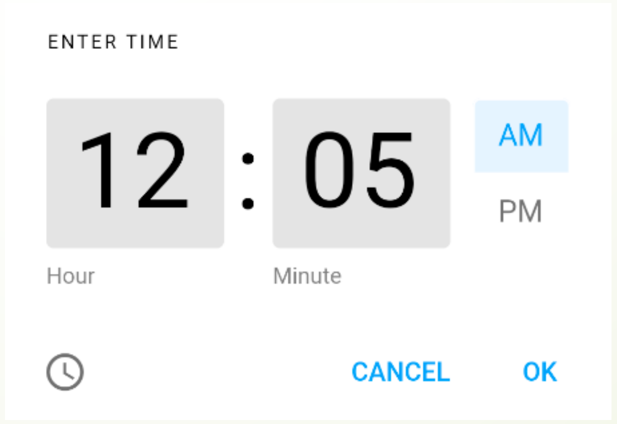
Date Picker

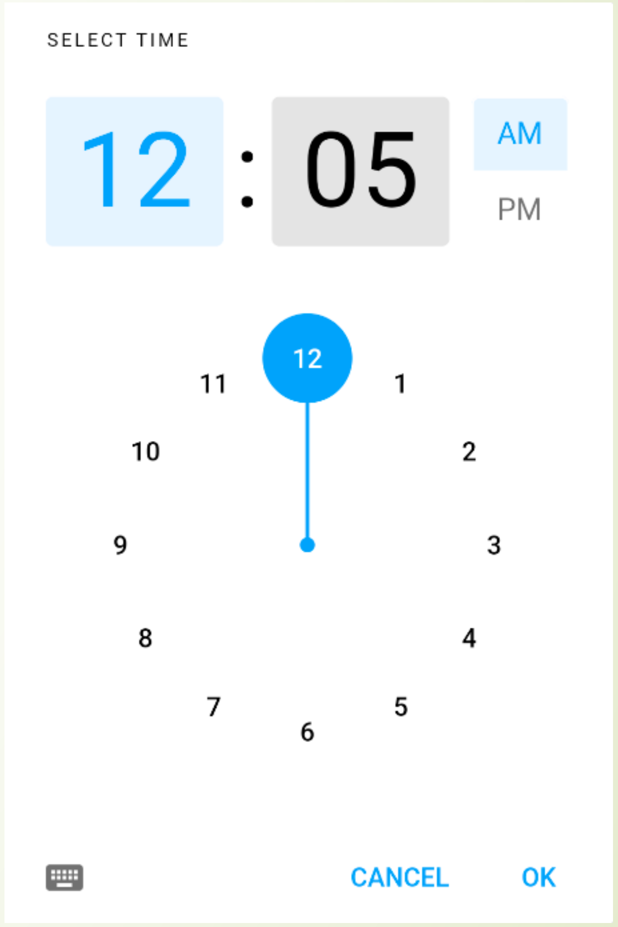
- The showDatePicker() method returns a Future<DateTime?> object, which represents a value that may not be available yet. When the user selects a date and taps the "OK" button, the Future completes with the selected date value.
- If the user cancels the picker by tapping the "CANCEL" button or by tapping outside the picker dialog, the Future completes with a null value.

```
DateTime? selectedDate = await showDatePicker(
    context: context,
    initialDate: DateTime.now(),
    firstDate: DateTime(2020),
    lastDate: DateTime(2025),
);
if (selectedDate != null) { print('${selectedDate.toString()}’); }
else {   print('Date picker was canceled.’);}
```

TimePicker Dialog

- Time Picker is a widget allowing users to select a time value from a predefined set of options. In Flutter, the time picker widget is a pre-built user interface element that can be easily added to your application's user interface.
- The time picker can be customized to suit your application's needs and can be used in a variety of contexts, from scheduling appointments to setting reminders.

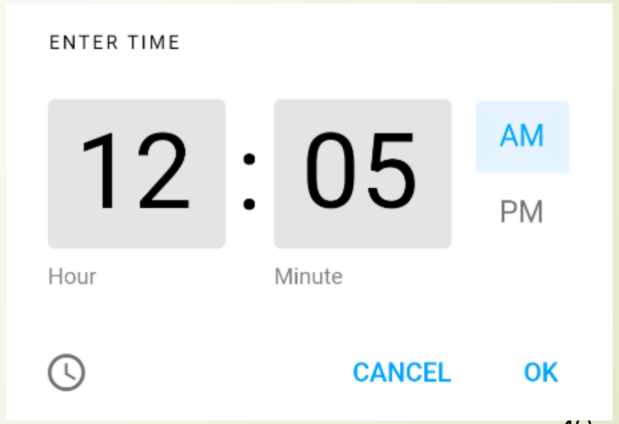




TimePicker Dialog

- The showTimePicker() method is a built-in Flutter function that displays a dialog box containing a time picker widget.
    - The context parameter is required and specifies the build context for the dialog box.
    - The initialTime parameter specifies the initial time value to be displayed in the time picker.
    - There are several other optional parameters that you can use to further customize the time picker's behavior. These parameters include useRootNavigator, routeSettings, helpText, cancelText, confirmText, initialEntryMode, initialDatePickerMode, and errorFormatText.

```
final TimeOfDay? picked = await showTimePicker(
    context: context,
    initialTime: _selectedTime,
);
```



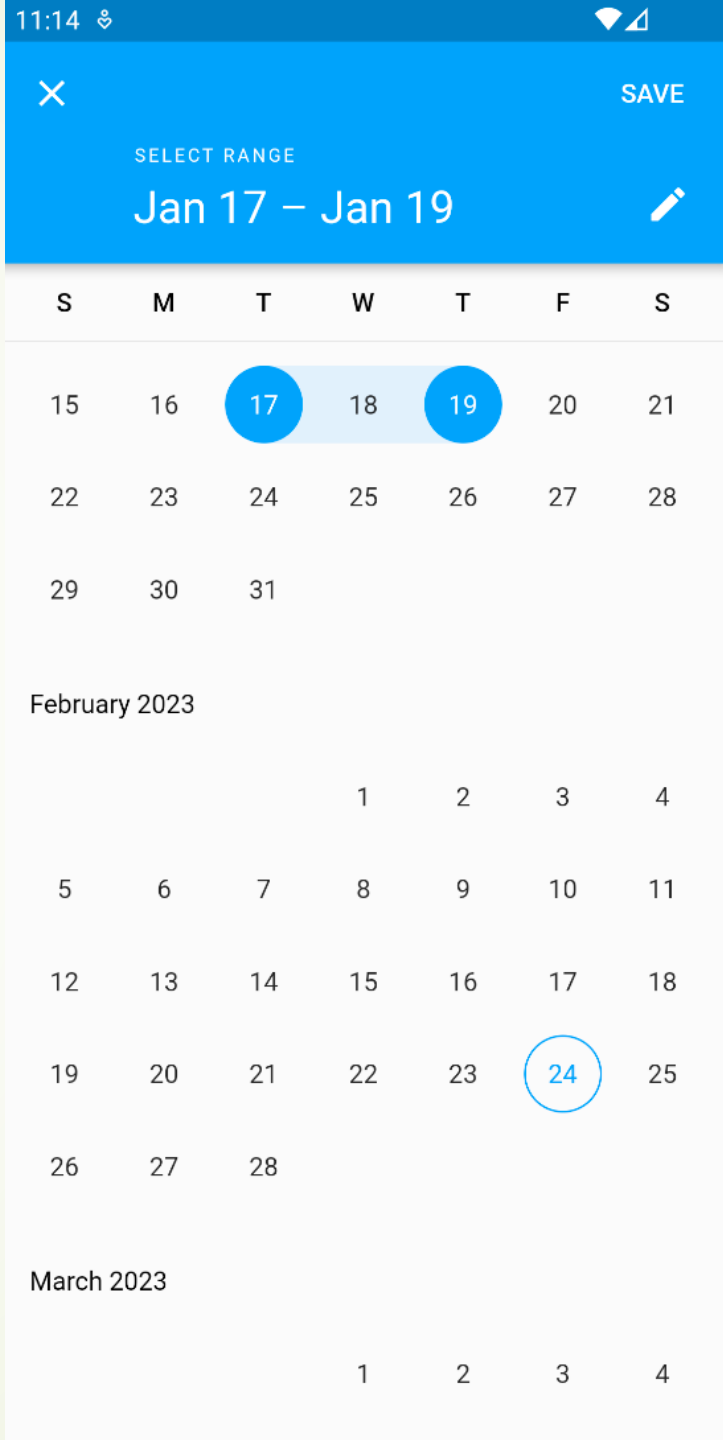
TimePicker Dialog

- When the user selects a time value in the time picker dialog box, the showTimePicker() method returns the selected time value as a Future<TimeOfDay> object.
- We then use the await keyword to wait for the user to select a time value, and store the selected value in a TimeOfDay? variable called pickedTime.

```
final TimeOfDay? picked = await showTimePicker(
    context: context,
    initialTime: _selectedTime,
);
if (picked != null && picked != _selectedTime) {
    setState(() {
        _selectedTime = picked;
    });
}
```

DateRange Picker

- The date range picker is a useful user interface component that allows users to select a range of dates from a calendar view.
- The date range picker widget can be used in a variety of contexts, such as for scheduling appointments or booking reservations.



DateRange Picker

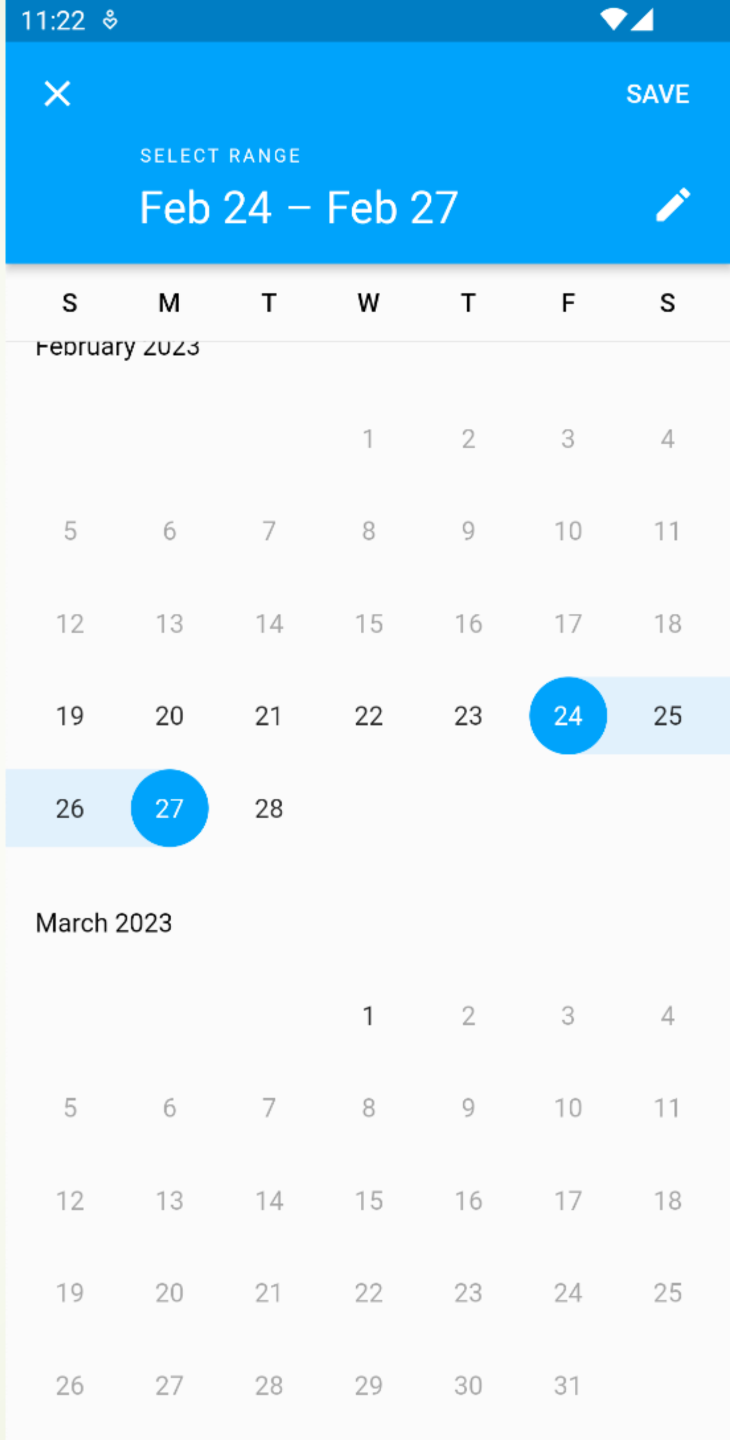
- showDateRangePicker is a built-in Flutter function that displays a date range picker dialog box to the user. It allows the user to select a range of dates using a calendar view, and returns the selected range as a DateTimeRange object.
- The function takes a number of parameters, such as context, firstDate, and lastDate, which specify the date range that the user can select. It also allows you to set an initialDateRange parameter to provide a default date range to display when the dialog box is first opened.

```
final pickedDateRange = await showDateRangePicker(
    context: context,
    firstDate: DateTime.now().subtract(Duration(days: 365)),
    lastDate: DateTime.now().add(Duration(days: 365)),
    initialDateRange: initialDateRange,
);
```

DateRange Picker

- To limit the start and end date of a date range picker in Flutter, you can use the firstDate and lastDate parameters when calling the showDateRangePicker function.

```dart
final initialDateRange = DateTimeRange(
          start: DateTime.now(),
          end: DateTime.now().add(Duration(days: 3)));

final pickedDateRange = await showDateRangePicker(
      context: context,
      firstDate: DateTime.now().subtract(Duration(days: 5)),
      lastDate: DateTime.now().add(Duration(days: 5)),
      initialDateRange: initialDateRange,
);
```



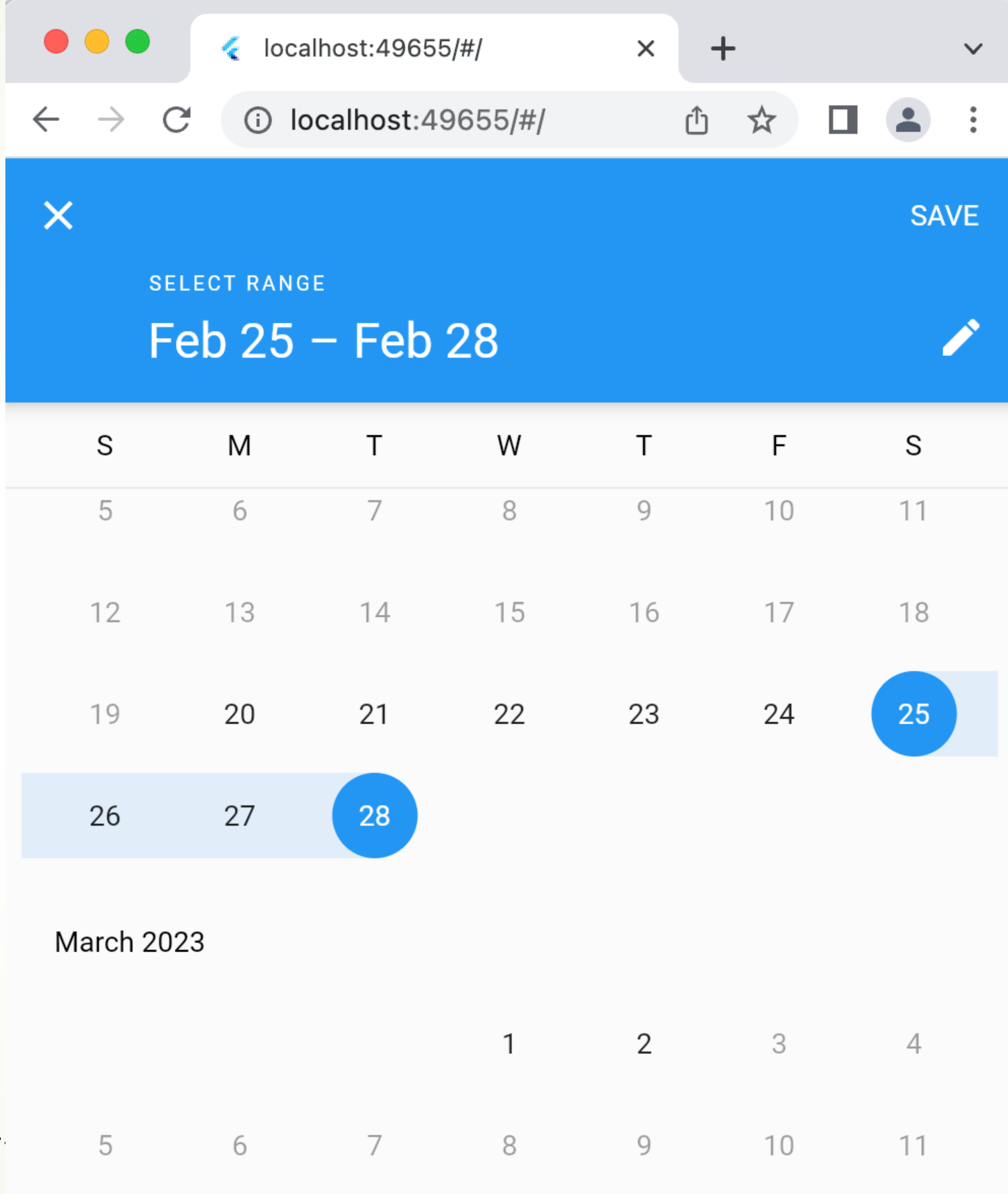
DateRange Picker

- Once the user has selected a range of dates, the showDateRangePicker function returns a Future<DateTimeRange?> object that represents the selected range. If the user cancels the dialog box or does not select a date range, the function returns null.

```
final pickedDateRange = await showDateRangePicker(…);

if (pickedDateRange != null && pickedDateRange != _selectedDateRange) {
    setState(() {
        // update UI here
    });
}
```

DateRange Picker

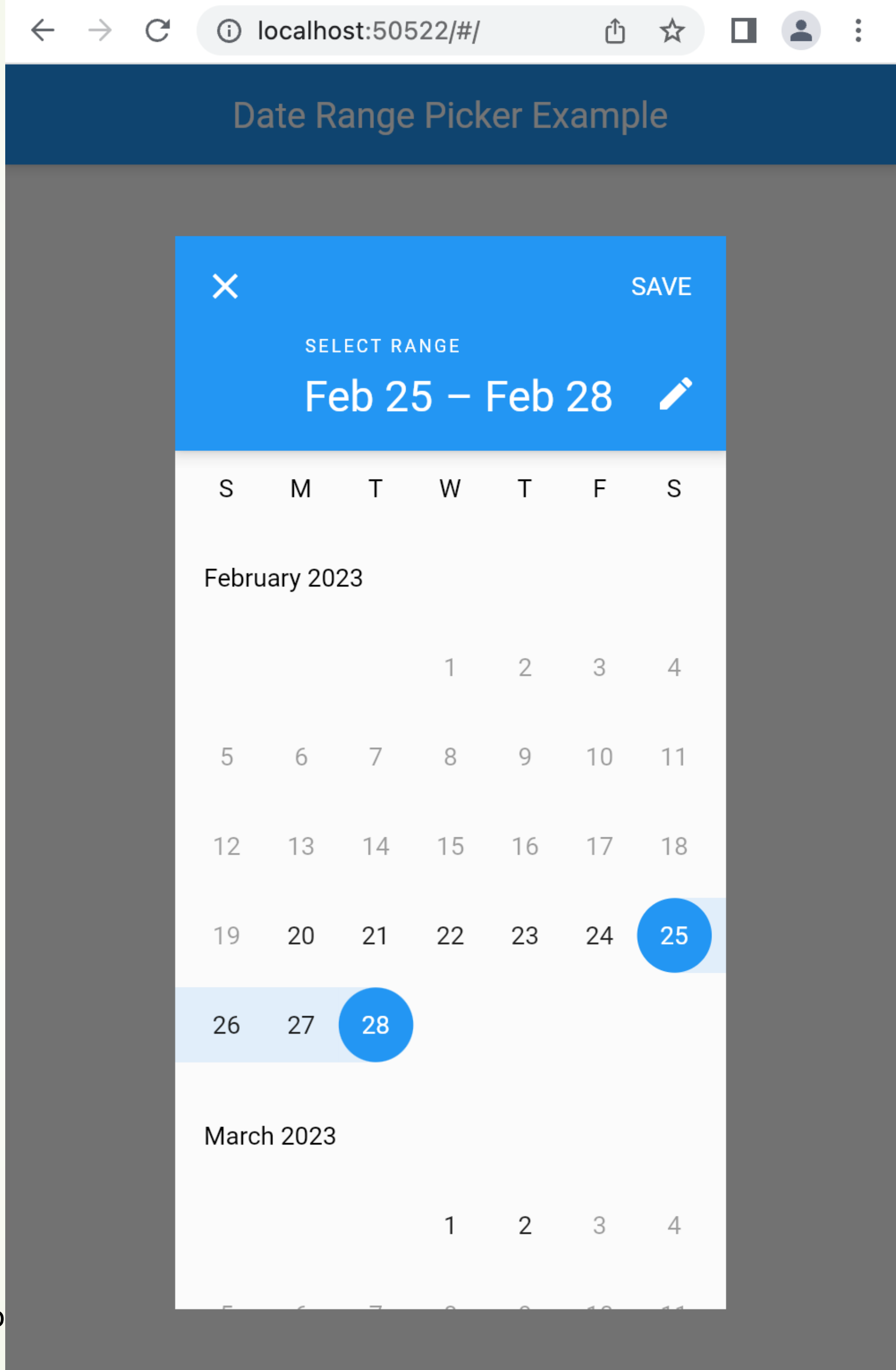
- By default, the date range picker dialog in Flutter web will take up the full screen, and the builder property can be used to customize the appearance and behavior of the date range picker.



DateRange Picker

- To limit the size of the dialog and prevent it from using the full screen, you can use the builder property to customize the appearance and behavior of the date range picker.

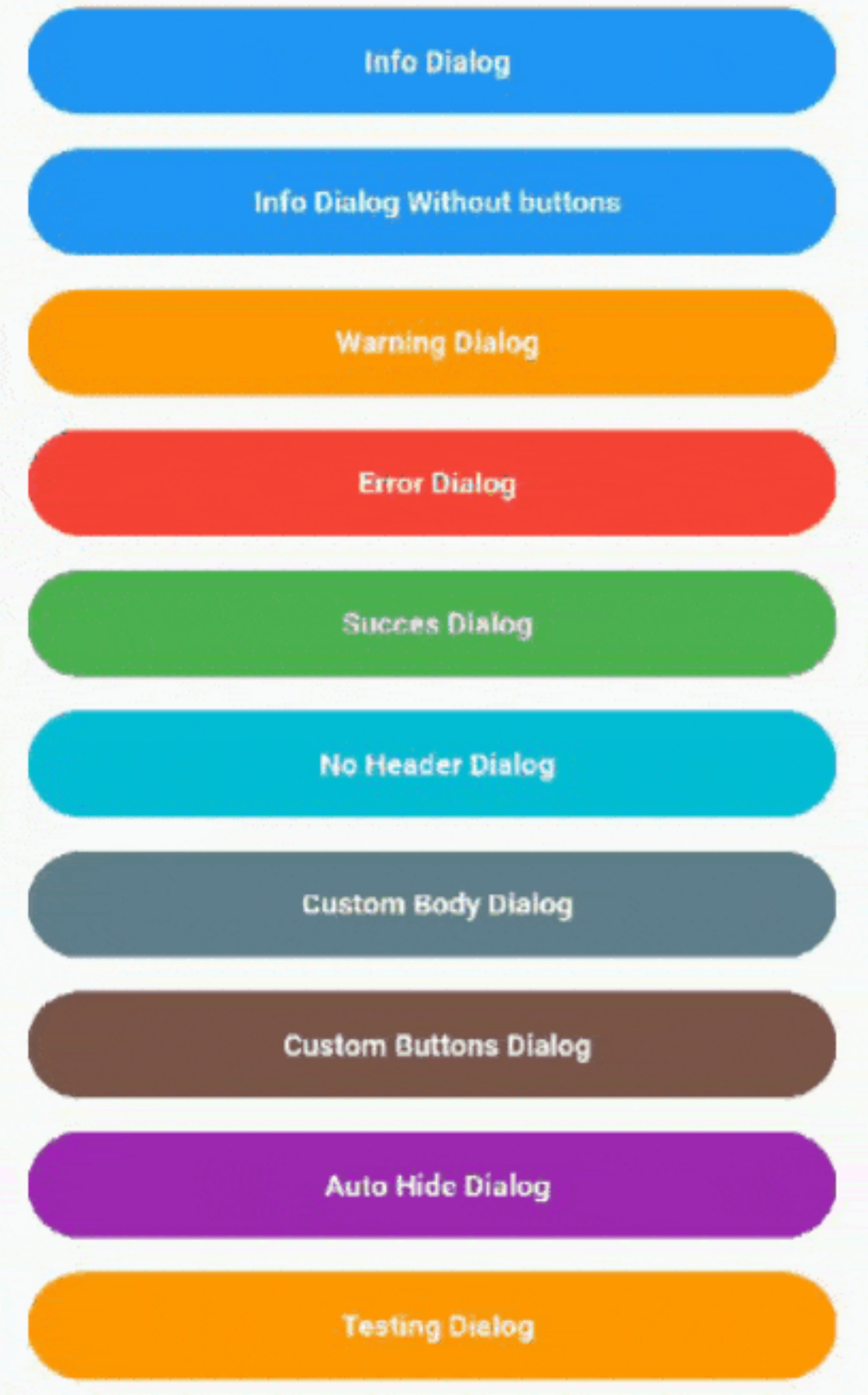
```
builder: (ctx, child) => Column(
    children: [
        ConstrainedBox(
            constraints: BoxConstraints(maxWidth: 400,
            maxHeight: 600),
            child: child,
        ),
    ],
),
```



Useful dialog packages

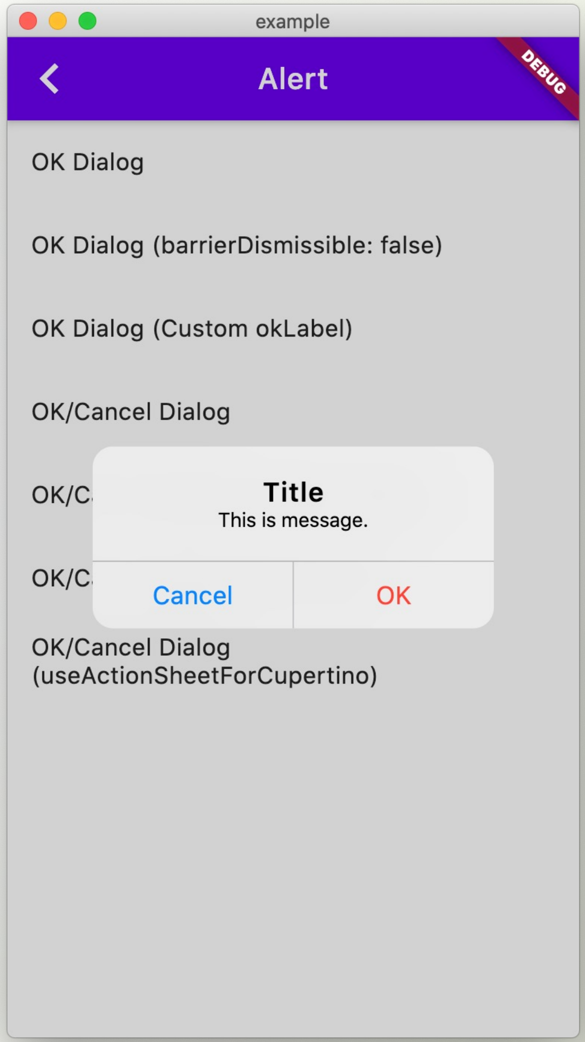
awesome_dialog

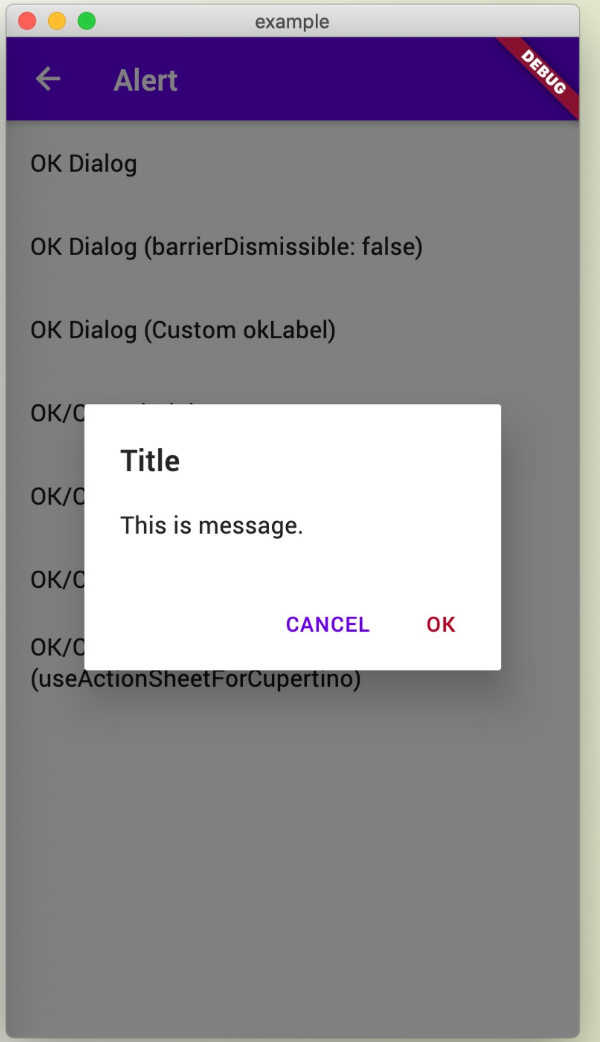
- A package that provides beautiful and customizable alert dialogs with multiple animation options.



adaptive_dialog

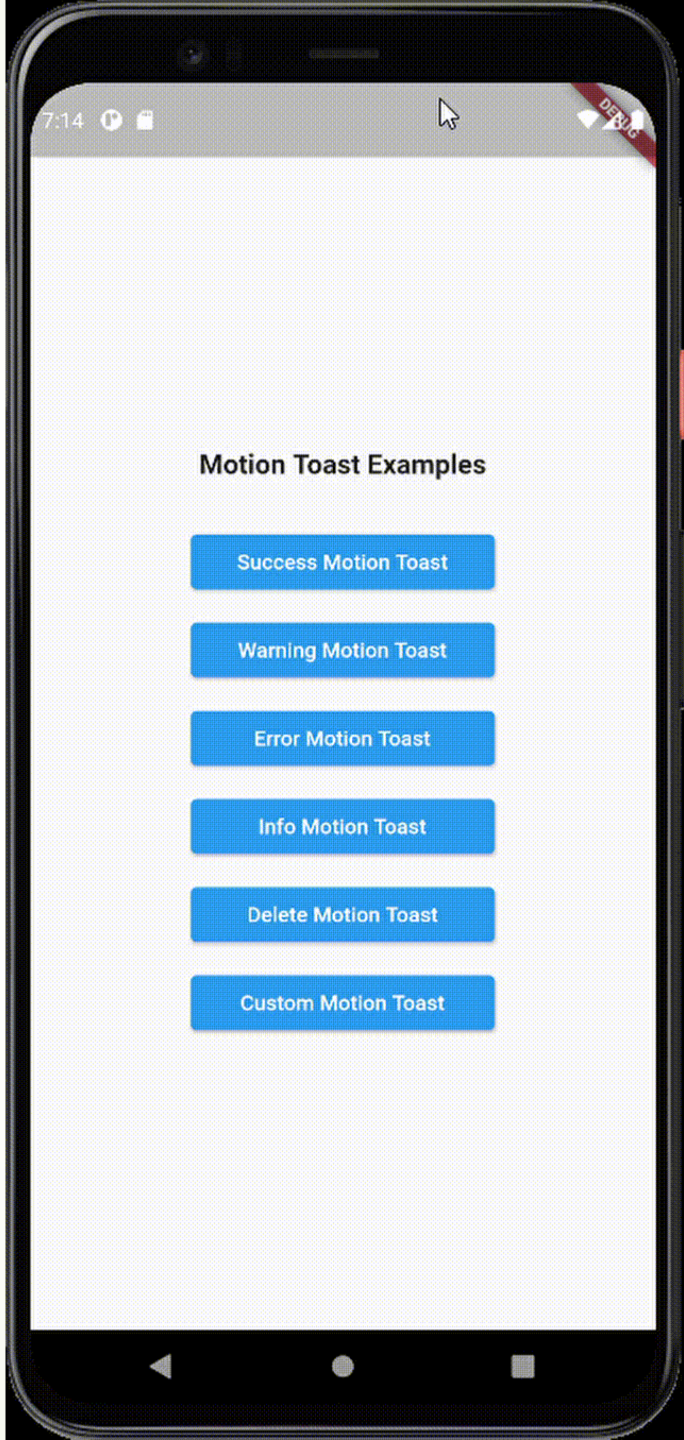
- The adaptive_dialog package is a powerful and highly customizable dialog package for Flutter that provides a set of adaptive dialogs that work across different platforms, including Android, iOS, and web.
- This package is designed to provide a consistent user experience across different platforms by automatically adapting to the platform's design guidelines and user interface conventions.

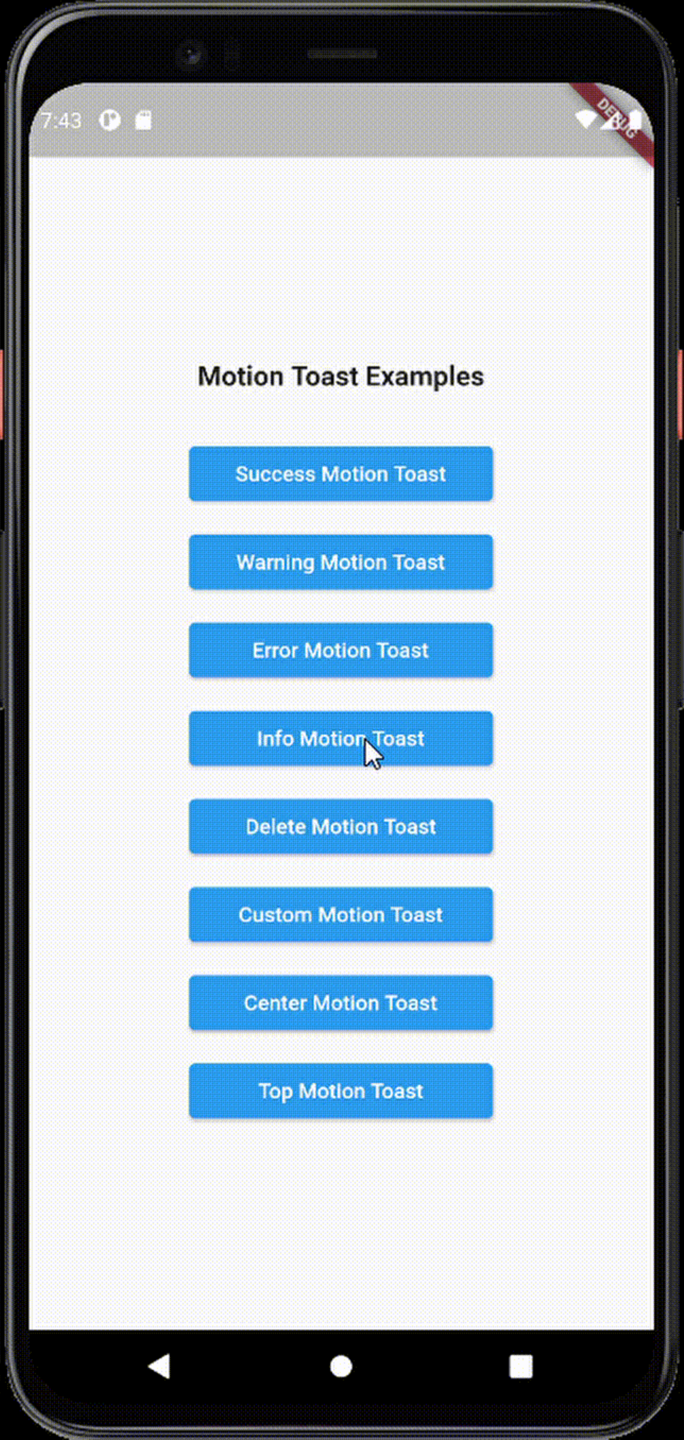

example
Alert
DEBUG
OK Dialog
OK Dialog (barrierDismissible: false)
OK Dialog (Custom okLabel)
OK/Cancel Dialog
OK/C
OK/C
OK/Cancel Dialog (useActionSheetForCupertino)
Title
This is message.
Cancel
OK


example
Alert
DEBUG
OK Dialog
OK Dialog (barrierDismissible: false)
OK Dialog (Custom okLabel)
OK/C
OK/C
OK/C
OK/C (useActionSheetForCupertino)
Title
This is message.
CANCEL
OK

motion_toast

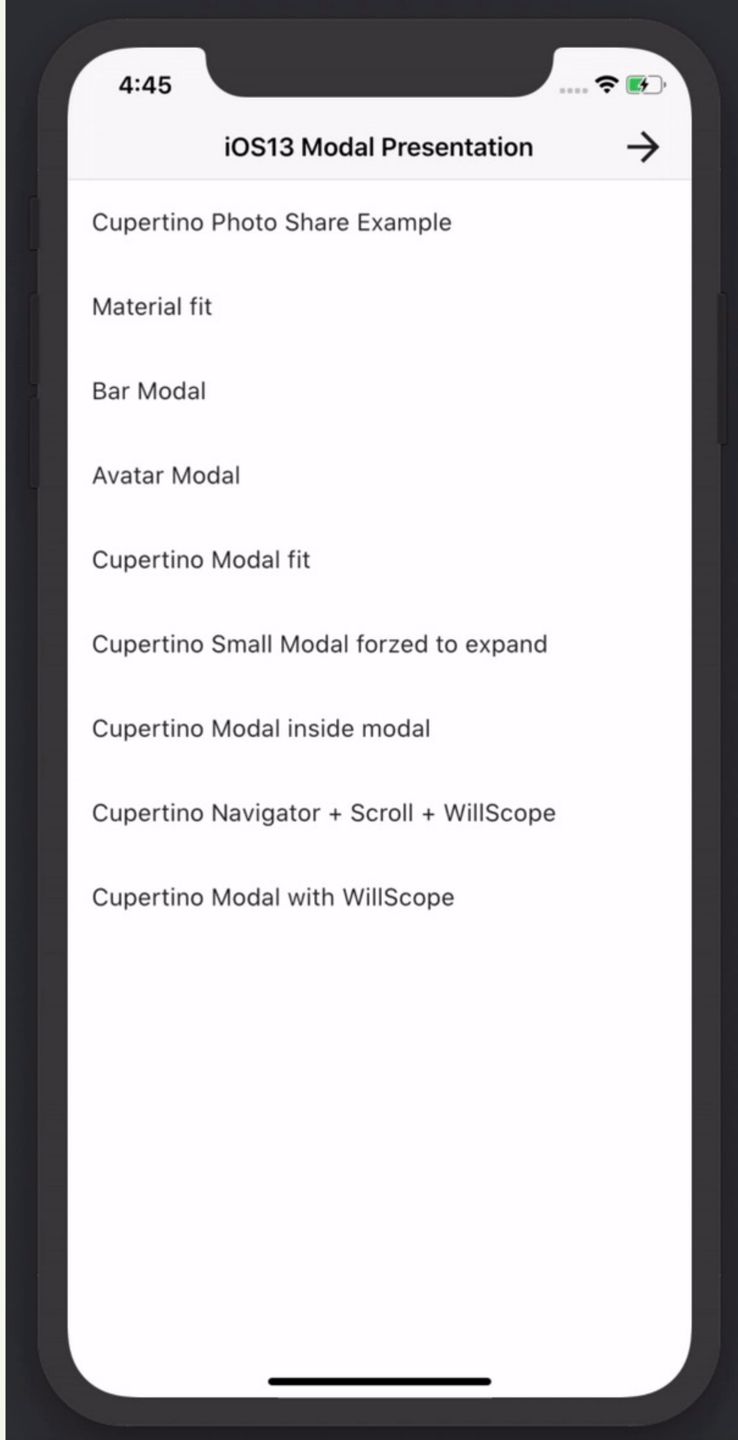
- A well designed toast with animations for all platforms Support material3 themes





modal_bottom_sheet

- Create awesome and powerful modal bottom sheets



4:45

iOS13 Modal Presentation

Cupertino Photo Share Example

Material fit

Bar Modal

Avatar Modal

Cupertino Modal fit

Cupertino Small Modal forzed to expand

Cupertino Modal inside modal

Cupertino Navigator + Scroll + WillScope

Cupertino Modal with WillScope

Working with Dialogs

- When working with dialogs in Flutter, there are a few things to keep in mind:
  - Context: Always remember to pass the context parameter when creating a dialog. The context parameter is essential as it provides information about the current state of the application and the widgets hierarchy.
  - Content: The content of the dialog should be clear and concise. Avoid displaying too much information at once as it may overwhelm the user.
  - Size: The size of the dialog should be appropriate for the content being displayed. It should not be too small that it is hard to read, nor too big that it takes up too much screen space.
  - User Experience: Dialogs should be used sparingly and only when necessary to avoid interrupting the user's workflow. Use appropriate animations and transitions to make the dialog appear and disappear smoothly.
  - Interaction: The user should be able to interact with the dialog in a clear and straightforward way. Use appropriate buttons and actions to allow the user to accept or reject the information presented in the dialog.