



502071

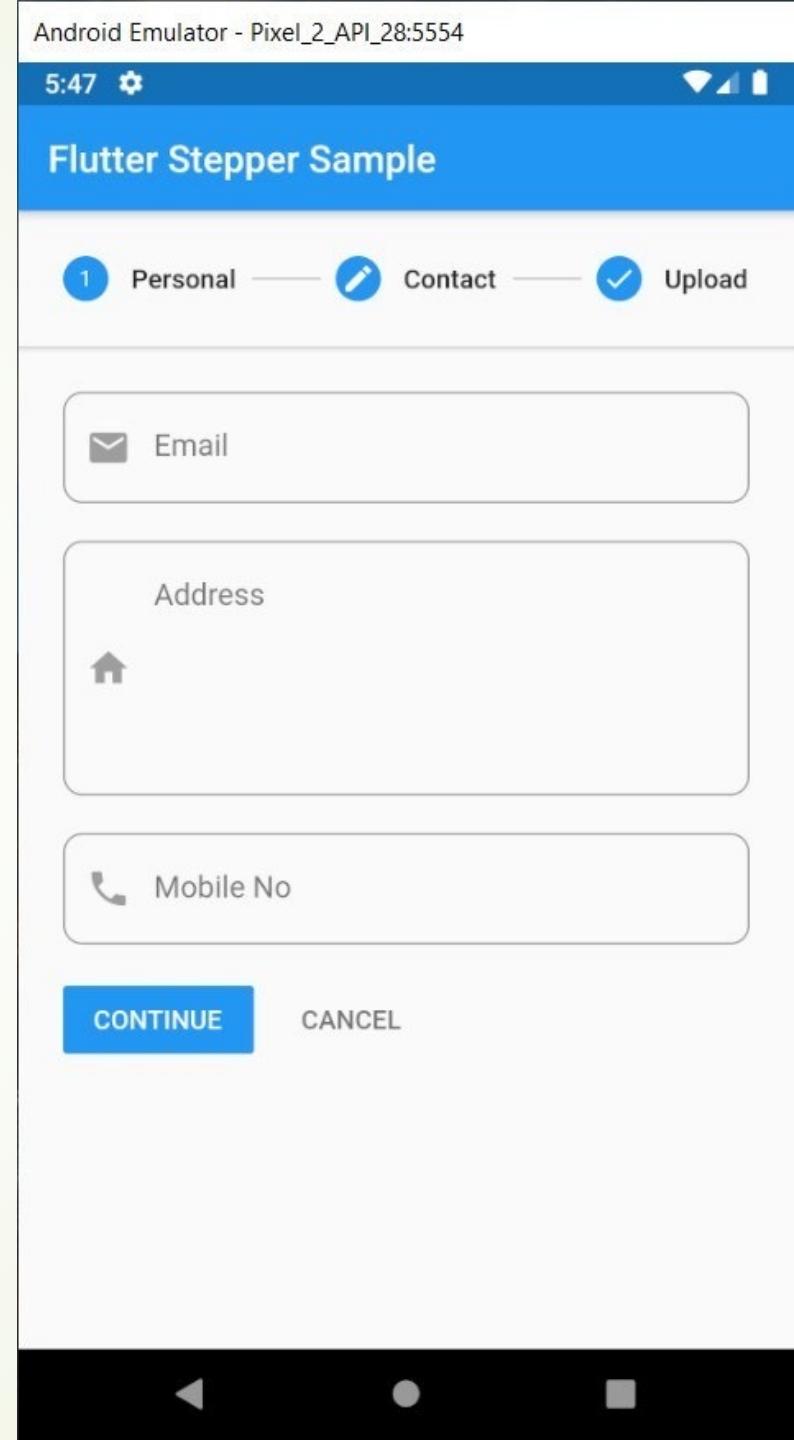
Cross-Platform Mobile Application Development

Working with Forms in Flutter

1

Introduction to Forms

- ▶ Forms are a common way for users to interact with an application, and they provide a way for users to communicate their needs, preferences, and intentions to the application.
- ▶ Forms are used in a variety of contexts, such as registration, login, checkout, and data entry. By providing a standardized set of input fields and labels, forms help users understand what information is required, and they guide users through the process of providing that information.
- ▶ Overall, forms are a crucial component of many user interfaces, and understanding how to design and implement effective forms is an important skill for developers and designers alike.



Flutter form-related widgets

► Flutter provides a variety of form-related widgets to help build and manage user input forms.

Some of these widgets include:

- **Form**: A widget that represents a form that contains form fields.
- **FormField**: A base class for form fields that can be used to create custom form fields.
- **TextFormField**: A form field that allows the user to enter text.
- **Checkbox**: A widget that allows the user to select a boolean value.
- **Switch**: A widget that allows the user to toggle a boolean value.
- **Radio**: A widget that allows the user to select a value from a group of options.
- **RadioListTile**: A form field that displays a list of options, each with a radio button.
- **DropdownButton**: A form field that displays a list of options, each in a dropdown menu.
- **DatePicker**: A form field that allows the user to select a date from a calendar.

TextField

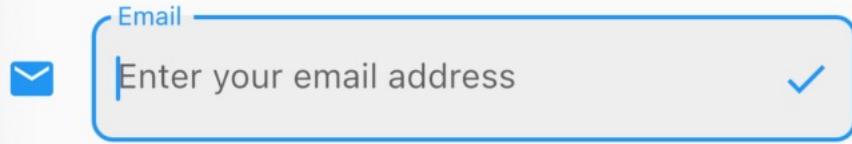
TextField widget

- ▶ TextField is a Flutter widget that allows users to enter and edit text. It is one of the most commonly used form-related widgets in Flutter and is used in a wide range of applications, such as search bars, login forms, and chat interfaces.
- ▶ To use TextField in your Flutter application, you can create a new instance of the widget and add it to your application's widget tree. Here's an example of how to create a basic TextField widget:

```
TextField(  
    decoration: InputDecoration(  
        labelText: 'Enter your name',  
    ),  
);
```

TextField widget

- ▶ You can also customize the behavior of TextField by setting a range of additional properties, such as `onChanged` to respond to changes in user input, `textInputAction` to configure the behavior of the keyboard, and `maxLines` to allow users to enter multiple lines of text.



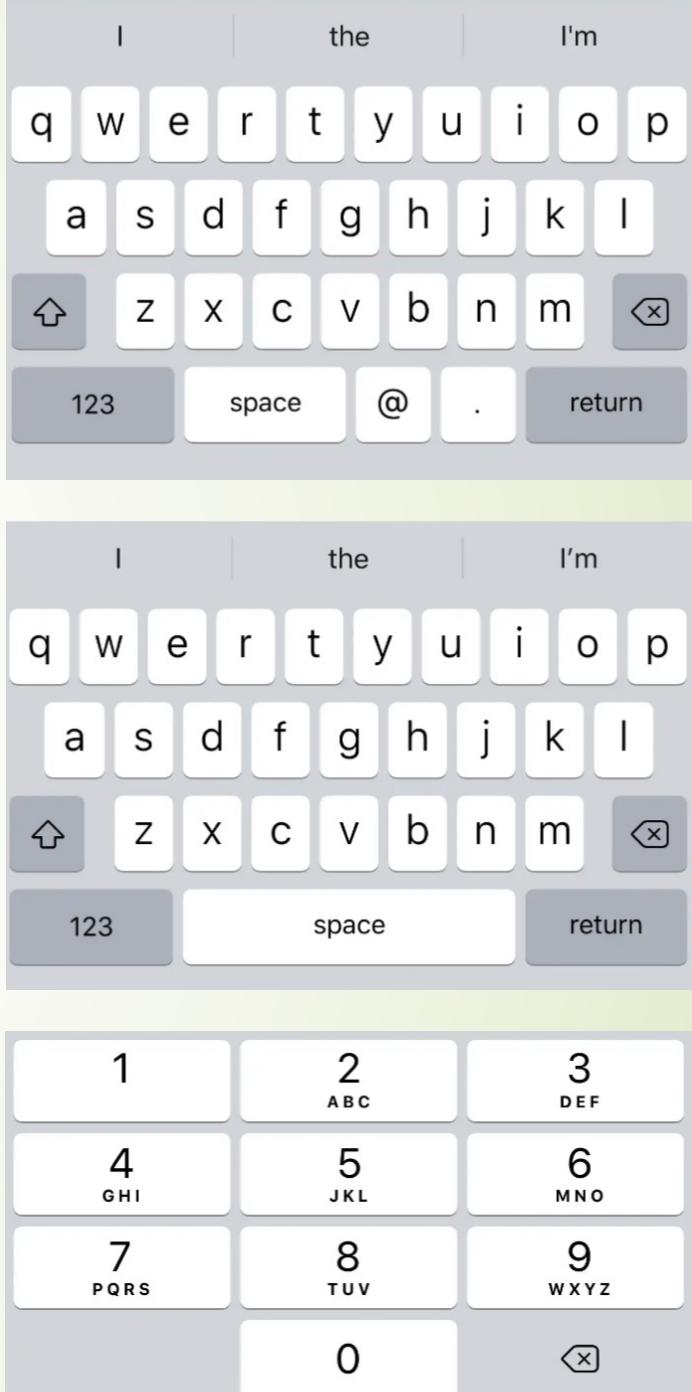
```
TextField(  
    onChanged: (text) { // Respond to changes in user input },  
    onSubmitted: (text) { // Handle text submission },  
    keyboardType: TextInputType.emailAddress,  
    textInputAction: TextInputAction.done,  
    maxLines: 3,  
    decoration: InputDecoration(  
        labelText: 'Enter your email',  
        hintText: 'e.g. example@domain.com',  
        border: OutlineInputBorder(),  
    ),  
);
```

TextField widget

- ▶ **onChanged** is a callback that is triggered whenever the user types or deletes a character in the TextField. This callback is useful for updating the state of your application in real-time as the user types, such as for validating user input or displaying a live preview of the entered text. The onChanged callback is triggered continuously as the user types, so it can be called multiple times for a single user action.
- ▶ **onSubmitted**, on the other hand, is a callback that is triggered when the user submits the entered text, typically by pressing the enter key on the keyboard. This callback is useful for handling the final submission of user input, such as for processing a search query or submitting a form. Unlike onChanged, onSubmitted is triggered only once per user action, when the user has finished entering the text.

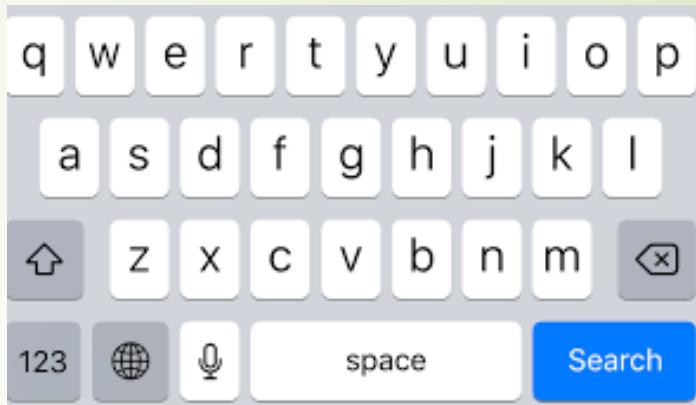
TextField widget

- ▶ `keyboardType` is an attribute that allows you to specify the type of keyboard that is displayed when the user interacts with the text field.
- ▶ Some of the most commonly used keyboard types include:
 - ▶ `TextInputType.text`: A standard keyboard for entering text.
 - ▶ `TextInputType.multiline`: A keyboard that allows the user to enter multiple lines of text.
 - ▶ `TextInputType.number`: A numeric keyboard for entering numbers.
 - ▶ `TextInputType.phone`: A numeric keyboard for entering phone numbers.
 - ▶ `TextInputType.datetime`: A keyboard for entering dates and times.
 - ▶ `TextInputType.emailAddress`: A keyboard for entering email addresses.



TextField widget

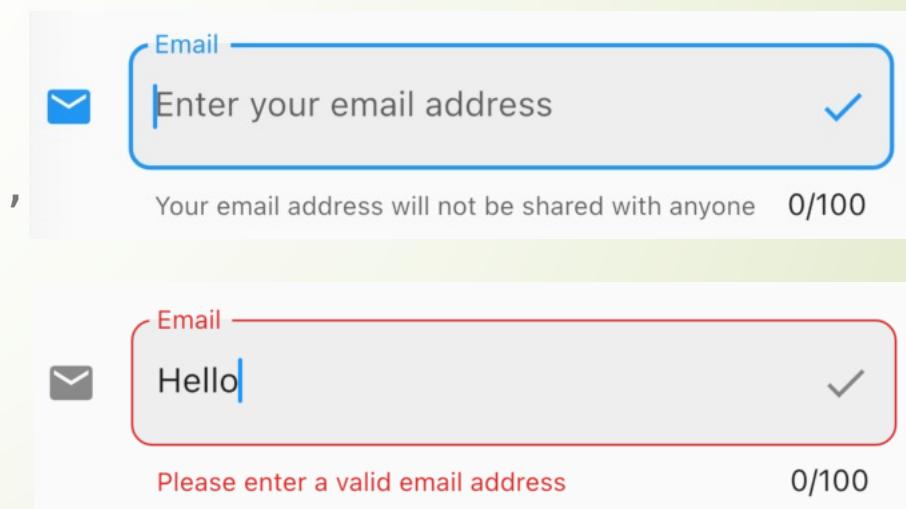
- ▶ `textInputAction` is an attribute that allows you to specify the action that should be performed when the user submits the text field's value, typically by tapping the "done" or "return" button on the on-screen keyboard.
- ▶ Some of the most commonly used actions include:
 - ▶ `TextInputAction.done`: The text input is complete and the input focus should be moved to the next field.
 - ▶ `TextInputAction.go`: The user wants to perform a "go" operation, such as submitting a search query.
 - ▶ `TextInputAction.send`: The user wants to perform a "send" operation, such as sending an email or chat message.
 - ▶ `TextInputAction.next`: The user wants to move the input focus to the next field without submitting the text input.
 - ▶ `TextInputAction.previous`: The user wants to move the input focus to the previous field without submitting the text input.



TextField widget

- ▶ `InputDecoration` is used to define the appearance of a text input field.

```
TextField(  
    decoration: InputDecoration(icon: Icon(Icons.email),  
    labelText: 'Email', hintText: 'Enter your email address',  
    helperText: 'Your email address will not be shared with anyone',  
    errorText: 'Please enter a valid email address',  
    suffixIcon: Icon(Icons.check), counter: Text('0/100'),  
    filled: true, fillColor: Colors.grey[200],  
    border: OutlineInputBorder(  
        borderRadius: BorderRadius.circular(10),  
        borderSide: BorderSide(color: Colors.grey),),  
    keyboardType: TextInputType.emailAddress,  
   textInputAction: TextInputAction.next,  
    maxLength: 100,  
)
```



Handle changes to a text field

- ▶ In some cases, it's useful to run a callback function every time the text in a text field changes. With Flutter, you have two options:
 - ▶ Supply an `onChanged()` callback to a `TextField` or a `TextFormField`.
 - ▶ Use a `TextEditingController`.
- ▶ The simplest approach is to supply an `onChanged()` callback to a `TextField` or a `TextFormField`. Whenever the text changes, the callback is invoked.

```
TextField(  
    onChanged: (text) {  
        print('First text field: $text');  
    },  
) ,
```

TextEditing Controller

TextEditingController

- ▶ TextEditingController is a class that provides a way to `read` and `modify` the text entered into a TextField or TextFormField. It allows you to programmatically modify the text value of the TextField.
- ▶ It also provides an `addListener` method that you can use to register a listener callback that is called whenever the text value changes.

TextEditingController

- To read the current value of a `TextField` in Flutter, you can use the `controller` property of the widget. First, create a `TextEditingController` and set it as the `controller` of your `TextField` widget. You can then access the current value of the `TextField` by reading the `text` property of the `TextEditingController`.

```
TextEditingController _controller = TextEditingController();  
TextField(  
    controller: _controller,  
    decoration: InputDecoration(  
        labelText: 'Name',  
        hintText: 'Enter your name',  
    ),  
);  
// To read the value of the TextField  
String name = _controller.text;
```

TextEditingController

- To set the value of a TextField, you can use the TextEditingController to set the value of the TextField.

```
TextEditingController _controller = TextEditingController();  
  
TextField(  
    controller: _controller,  
    decoration: InputDecoration(  
        labelText: 'Name',  
        hintText: 'Enter your name',  
    ),  
);  
  
// To set the value of the TextField  
_controller.text = 'John Doe';
```

TextEditingController

- ▶ It also provides an `addListener` method that you can use to register a listener callback that is called whenever the text value changes.

```
class _MyAppState extends State<MyApp> {  
  var _controller = TextEditingController();  
  
  @override  
  void initState() {  
    _controller.addListener(() {  
      print('New text value is ${_controller.text}');  
    });  
    super.initState();  
  }  
  ...  
}
```

TextEditingController

- The controller should be cleaned up when the widget is removed from the widget tree

```
class _MyAppState extends State<MyApp> {  
  var _controller = TextEditingController();  
  
  @override  
  void dispose() {  
    _controller.dispose();  
    super.dispose();  
  }  
  
  ...  
}
```

Focus Node

- ▶ To give focus to a `TextField` in Flutter programmatically, you can use a `FocusNode` object to manage the focus state of the widget.
- ▶ First, create a `FocusNode` object and set it as the `focusNode` property of your `TextField` widget. Then, to give focus to the `TextField`, call the `requestFocus()` method on the `FocusNode` object.

```
FocusNode _focusNode = FocusNode();  
  
TextField(  
    focusNode: _focusNode,  
    decoration: InputDecoration(  
        labelText: 'Name',  
        hintText: 'Enter your name',  
    ),  
);  
  
// To give focus to the TextField  
_focusNode.requestFocus();
```

Working with TextField widget

- ▶ When working with the TextField widget in Flutter, there are a few things you should keep in mind:
 - ▶ **Performance:** TextField widgets can be expensive to render, especially if you have a lot of them on the screen. Consider using the ListView.builder widget so that only the widgets that are currently visible on the screen are rendered.
 - ▶ **Validation:** It's important to validate user input in your TextField widgets to ensure that the data is in the correct format.
 - ▶ **Text Input:** You should use the keyboardType andtextInputAction properties to ensure that the user is presented with the correct type of keyboard for the input field and that the correct action is triggered when the user submits the form.

TextField

TextField widget

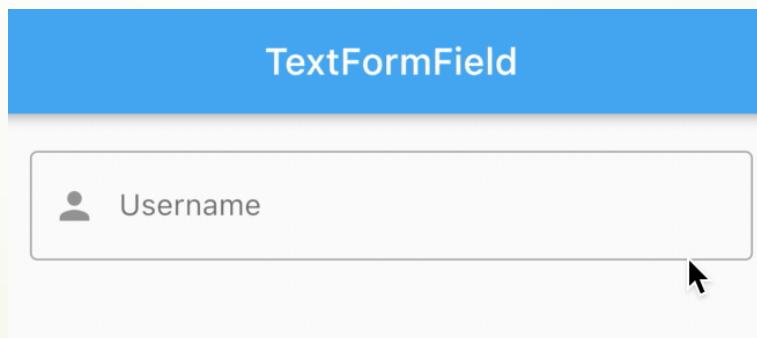
- ▶ TextField is a basic widget that provides a user interface for entering text. It has a single line by default, and it allows the user to enter a single line of text.
- ▶ TextFormField, on the other hand, is a more advanced widget that provides additional features such as validation and form submission. It is built on top of TextField and provides a way to validate the user input and display an error message if the input is invalid. It also provides a way to submit the form when the user is done entering data.

```
TextFormField(  
    validator: (value) {  
        // return "Error message if there is an error"  
        return null; // otherwise  
    },  
    onFieldSubmitted: (value) { // Do something when the user submits the form},  
)
```

TextField widget

- To customize a TextField widget in Flutter, you can use the `InputDecoration` class to define the look and feel of the text input field.

```
TextField(  
  decoration: InputDecoration(  
    labelText: 'Username',  
    hintText: 'Enter your username',  
    prefixIcon: Icon(Icons.person),  
    border: OutlineInputBorder(),  
    focusedBorder: OutlineInputBorder(  
      borderSide: BorderSide(color: Colors.blue, width: 2.0),  
    ),  
  ),  
,  
);
```



TextField widget

- To show an error message and error border for a TextField widget in Flutter, you can use the `errorText` and `errorBorder` properties of the `InputDecoration` class.

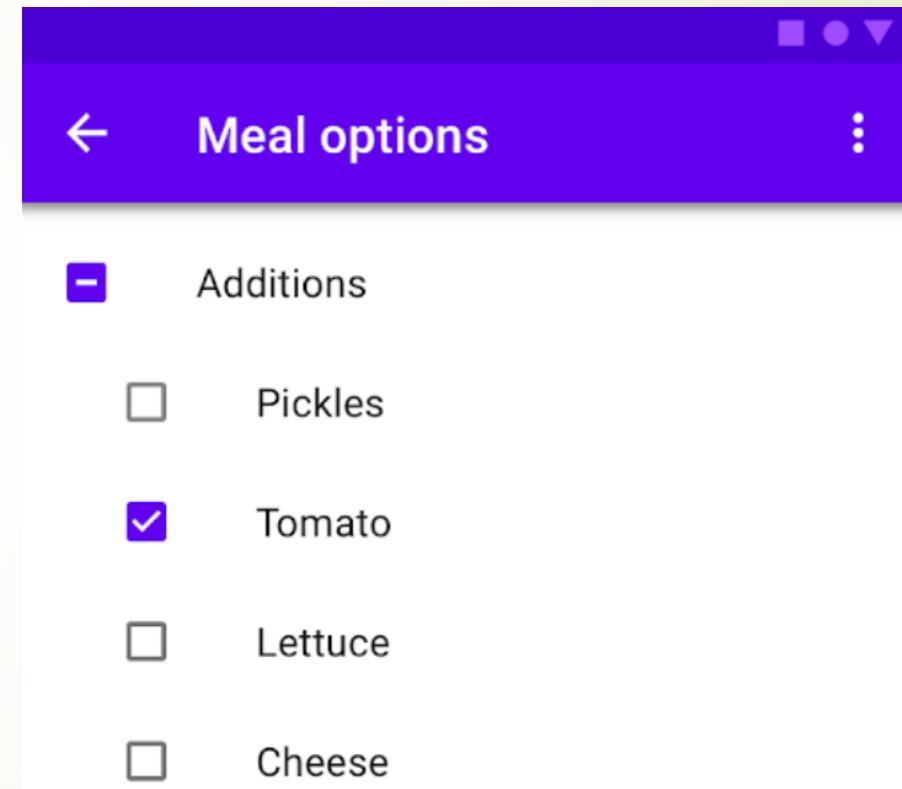
TextField(

```
    decoration: InputDecoration(  
        labelText: 'Username',  
        hintText: 'Enter your username',  
        prefixIcon: Icon(Icons.person),  
        border: OutlineInputBorder(),  
        errorText: 'Please provide a valid username',  
        errorBorder: OutlineInputBorder(  
            borderSide: BorderSide(color: Colors.red, width: 2.0),  
        ),  
        focusedBorder: OutlineInputBorder(  
            borderSide: BorderSide(color: Colors.blue, width: 2.0),  
        ),  
    ),  
);
```

Checkbox

Checkbox widget

- Flutter Checkbox widget is a material design widget used for enabling a user to select one or more options from a set of options. It provides a checkbox with a label to display the description of the option.
- The state of the checkbox is maintained by a **boolean variable** that changes its value when the user interacts with the widget.



Checkbox widget

- To create a Checkbox widget in Flutter, you can use the Checkbox class. It requires a boolean value to represent the state of the checkbox, which can be updated using the `onChanged` callback function.

```
bool _isChecked = false;  
  
Checkbox(  
    value: _isChecked,  
    onChanged: (bool value) {  
        setState(() {  
            _isChecked = value;  
        });  
    },  
) ,
```



Checkbox widget

- To set a text label for the Flutter Checkbox widget, you can wrap the Checkbox widget inside a Text widget. Here is an example code snippet:

```
Row(  
    children: [  
        Checkbox(  
            value: _isChecked,  
            onChanged: (bool? value) {  
                setState(() {  
                    _isChecked = value;  
                });  
            },  
        ),  
        Text('I agree to the terms and conditions.'),  
    ],  
)
```

I agree to the terms and conditions.

Checkbox widget

- ▶ You can customize the appearance of the Checkbox widget by using the properties available in the Checkbox class. For example, you can change the color of the checkbox using the `activeColor` property, and the color of the label using the `textColor` property.

```
Checkbox(  
    activeColor: Colors.green,  
    checkColor: Colors.white,  
    shape: RoundedRectangleBorder(  
        borderRadius: BorderRadius.circular(5.0),  
    ),  
    value: _isChecked,  
    onChanged: (bool? value) {  
        setState(() {  
            _isChecked = value;  
        });  
    },  
)
```



I agree to the terms and conditions.

Handling Checkbox Groups

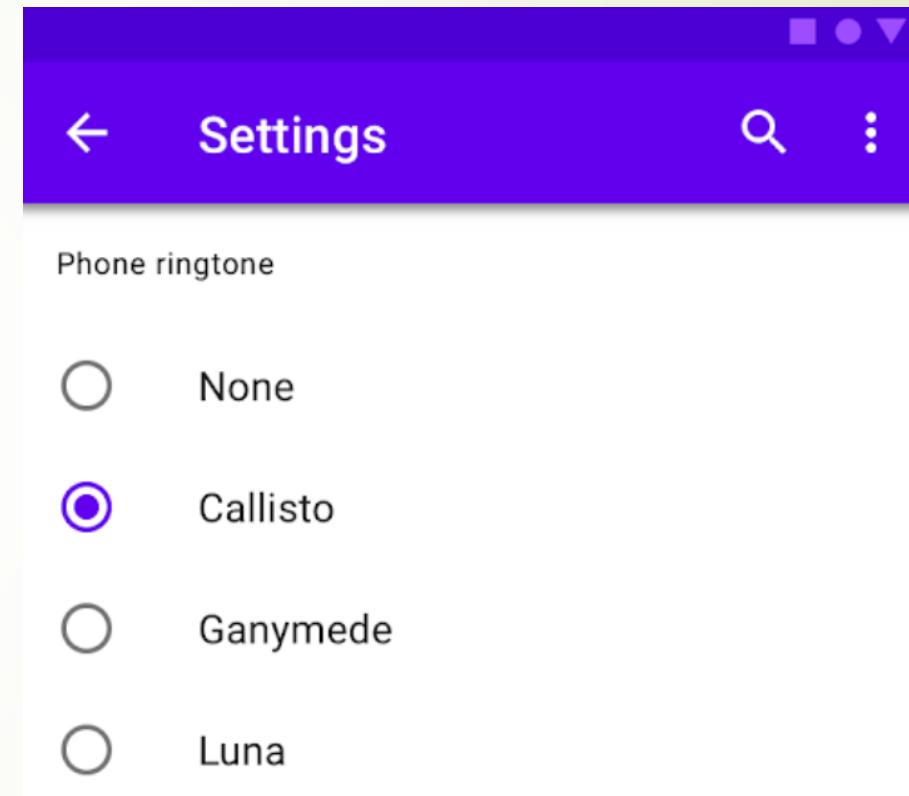
- To handle a group of checkboxes, you can use a `CheckboxListTile` widget. It provides a `label` for the checkbox, and also enables the user to select multiple options from a set of options.

```
List<bool?> _isCheckedList = [false, false, false];  
Column( children: [  
    CheckboxListTile( value: _isCheckedList[0],  
        title: Text("Option 1"),  
        onChanged: (bool value) {  
            setState(() { _isCheckedList[0] = value; });  
        },  
    ),  
    CheckboxListTile( value: _isCheckedList[1],  
        title: Text("Option 2"),  
        onChanged: (bool value) {  
            setState(() { _isCheckedList[1] = value; });  
        },  
    ),  
    CheckboxListTile( value: _isCheckedList[2],  
        title: Text("Option 3"),  
        onChanged: (bool value) {  
            setState(() { _isCheckedList[2] = value; });  
        },  
    ),  
> )
```

Radio Button

RadioButton widget

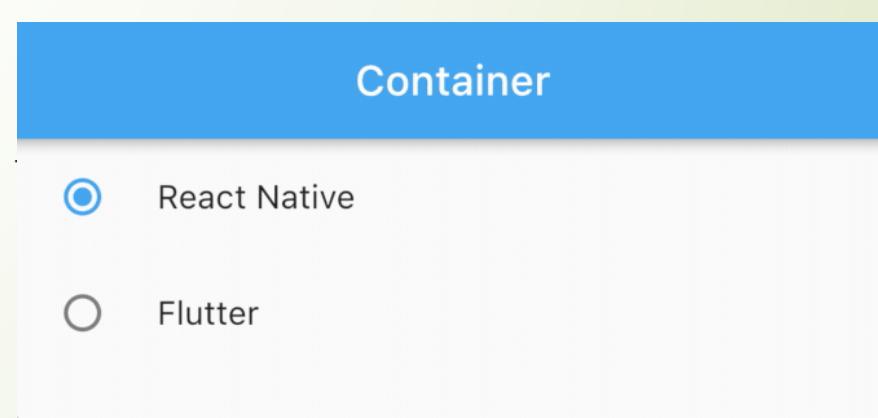
- ▶ RadioButton widget is used for enabling a user to select one option from a set of options. It provides a circular radio button with a label to display the description of the option. The user can select a radio button by tapping on it, and the state of the radio button can be updated programmatically.



RadioButton widget

- To create a group of radio buttons, you can use the `RadioListTile` widget. The `RadioListTile` widget provides a title for the radio button. You can create multiple `RadioListTile` widgets with the `same groupValue` property to create a group of radio buttons. The `groupValue` property is used to maintain the state of the radio buttons in the group.

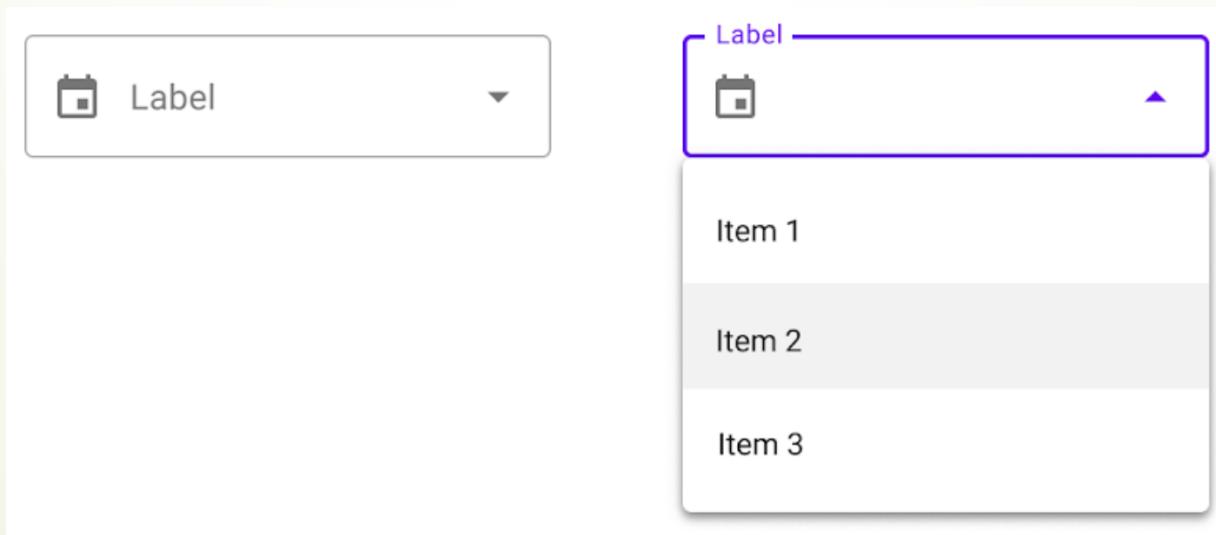
```
int? _radioValue;  
  
RadioListTile(title: const Text('React Native'),  
    value: 1, groupValue: _radioValue, onChanged: (value) {  
        setState(() { _radioValue = value; });  
    },  
,  
RadioListTile(title: const Text('Flutter'),  
    value: 2, groupValue: _radioValue, onChanged: (value)  
        setState(() {_radioValue = value;}));  
,  
)
```



Dropdown Button

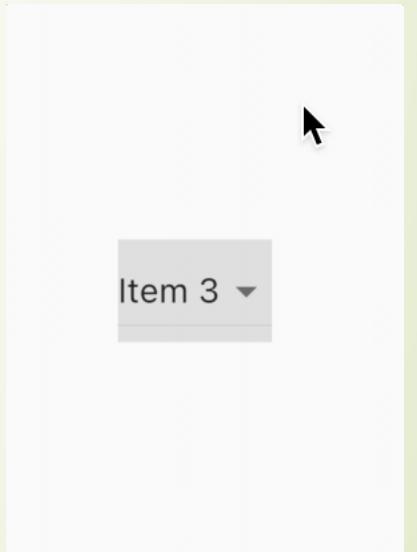
Dropdown Button widget

- ▶ The `DropdownButton` widget in Flutter is a common UI element used to display a list of items in a drop-down menu. It allows the user to select an item from a list of pre-defined options, similar to the HTML `select` tag or the native Android `Spinner` widget.
- ▶ To use the `DropdownButton` widget, you need to define a list of items to be displayed in the drop-down menu. Each item in the list should be represented by a `DropdownMenuItem` widget, which consists of a value and a child.



Dropdown Button widget

```
List<String> data = ['Item 1', 'Item 2','Item 3',];  
String? selectedItem = 'Item 1';  
  
late List<DropdownMenuItem<String>> dropdownItems = data.map((item) =>  
    DropdownMenuItem(value: item, child: Text(item))).toList();  
  
DropdownButton(value: selectedItem, items: dropdownItems,  
onChanged: (value) {  
    setState(() {  
        selectedItem = value;  
    });  
},  
)
```



Dropdown Button widget

- ▶ You can customize the text style of the selected item and the dropdown icon using the `style` and `icon` properties, respectively.

```
DropdownButton(  
    value: selectedItem,  
    items: dropdownItems,  
    style: TextStyle(fontSize: 18, color: Colors.blue),  
    icon: Icon(Icons.arrow_drop_down),  
    onChanged: (value) {  
        setState(() {  
            selectedItem = value;  
        });  
    },  
)
```



DropdownButtonFormField widget

- ▶ DropdownButtonFormField is designed to be used in a Form widget and it automatically manages the state of the dropdown. When a user selects an option, the form is validated, and the dropdown value is updated. This makes it easy to use in a form, as it manages the state for you. Additionally, it can be easily styled using the decoration property to match the style of the form.
- ▶ On the other hand, DropdownButton is more flexible and can be used in any widget tree, not just in a form. It does not manage the state of the dropdown, so the developer must manage the state themselves. This can make it more complex to use, but it also provides more control over the behavior of the dropdown.

The image shows a mobile application's registration screen. At the top is a navigation bar with three icons. Below the navigation bar is a header section with a logo and the text "Create account". The main content area contains the following fields:

- A text input field labeled "Email".
- A text input field labeled "Age".
- A dropdown menu labeled "Country" with "Cambodia" selected. A small blue line indicates the dropdown is open.
- A text input field labeled "Password".
- A large blue button at the bottom labeled "Register".

The entire form is set against a light green background with abstract brown and grey curved lines on the left side.

DropdownButtonFormField widget

- ▶ Here are some important attributes of the DropdownButtonFormField widget:
 - ▶ **value**: This attribute holds the currently selected value in the dropdown. It should be of the same type as the values of the items list.
 - ▶ **items**: This attribute is a list of DropdownMenuItem objects that represent the options available in the dropdown. Each DropdownMenuItem has a value and a child widget that is used to display the option in the dropdown.
 - ▶ **onChanged**: This attribute is a callback function that is called when the user selects an option from the dropdown. It should take a single argument that is the new value of the dropdown.
 - ▶ **decoration**: This attribute is used to apply decoration to the DropdownButtonFormField. It can be used to customize the border, label, hint, and other aspects of the widget.
 - ▶ **validator**: This attribute is a function that is called to validate the value of the dropdown when the form is submitted. It should return an error message as a string if the value is not valid, and null if the value is valid.
 - ▶ **onSaved**: This attribute is a callback function that is called when the form is saved. It should take a single argument that is the current value of the dropdown and can be used to update the form data.

DropdownButtonFormField widget

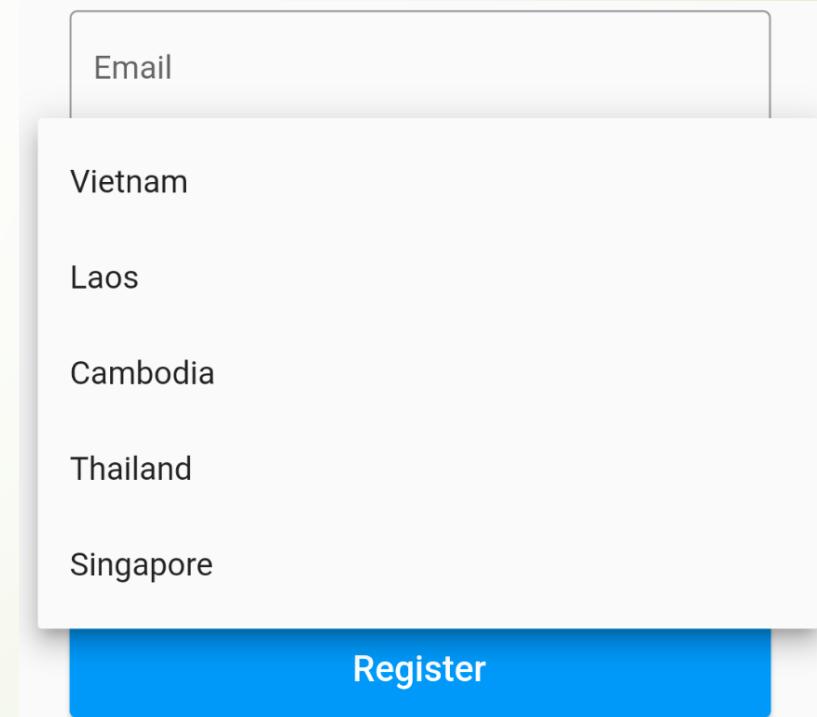
```
DropdownButtonFormField(isExpanded: true,  
    value: selected, decoration: InputDecoration(  
        labelText: 'Country',  
        border: OutlineInputBorder()),  
    items: countries.map((e) => DropdownMenuItem(value: e, child: Text(e))).toList(),  
    onChanged: (v) => {  
        setState(() {  
            selected = v!;  
        });  
    })  
)
```

Age

Country

Vietnam

Password



Handling Form

Handling Form

- ▶ In Flutter, handling forms involves managing user input, validating it, and processing the form data. You can create forms in Flutter using the Form widget and its child widgets such as TextFormField and DropdownButtonFormField.

```
final _formKey = GlobalKey<FormState>();  
String userName = '';  
  
Form(key: _formKey,  
      child: Column(children: <Widget>[  
          // Add form fields here  
      ],),  
);
```

- ▶ Here, we're creating a Form widget and assigning a [GlobalKey](#) to it. This key is used to [access the form's state](#) from within the widget tree.

Handling Form

- ▶ Add form fields:

- ▶ You can add form fields inside the Form widget using child widgets such as TextFormField, DropdownButtonFormField, etc. Here's an example of how to add a TextFormField:

```
TextFormField(  
    decoration: const InputDecoration(labelText: 'Name',),  
    validator: (value) {  
        if (value?.isEmpty ?? true) {  
            return 'Please enter your name';  
        }  
        return null;  
    },  
    onSaved: (value) { userName = value; },  
)
```

Handling Form

- The `validator` takes a function that receives the input value and `returns an error message` if the input is invalid, or `null` if it is valid.

```
TextField(  
    validator: (value) {  
        if (value == null || value.isEmpty) {  
            return 'Please enter your name';  
        }  
        return null;  
    }  
)
```

Handling Form

- ▶ Validate the form

- ▶ To validate the form, you can call the `validate()` method of the `FormState` object. Here's an example of how to validate the form:

```
void _submitForm() {  
  if (_formKey.currentState?.validate() ?? false) {  
    _formKey.currentState?.save();  
    // Process the form data  
  }  
}
```

- ▶ Here, we're calling the `validate()` method of the form's `GlobalKey`, which returns a boolean indicating whether the form is valid or not. If the form is valid, we're calling the `save()` method of the `FormState` object to save the form data.

Handling Form

- ▶ Process the form data

- ▶ After the form is validated and the data is saved, you can process the form data. Here's an example of how to process the form data:

```
void _submitForm() {  
    if (_formKey.currentState?.validate() ?? false) {  
        _formKey.currentState?.save();  
        // Process the form data  
        print(userName);  
    }  
}
```

- ▶ Here, we're printing the form data to the console. You can process the form data in any way you like, such as sending it to a server or saving it to a local database.

Handling Form

▶ Reset the form

- ▶ After the form is submitted, you may want to clear the form fields and **reset the form** to its initial state.

Here's an example of how to reset the form:

```
void _submitForm() {  
    if (_formKey.currentState?.validate() ?? false) {  
        _formKey.currentState?.save();  
        // Process the form data  
        print(userName);  
        _formKey.currentState.reset();  
    }  
}
```

- ▶ By following these steps, you can easily handle forms in Flutter and process user input. You can customize the form fields and add more validation rules to suit your needs.

Soft Keyboard

Soft Keyboard

- ▶ It's a common issue that the soft keyboard can sometimes cause the `TextField` widget to be hidden when it is displayed. This can be frustrating for users and can lead to a poor user experience.
- ▶ The reason for this issue is that the soft keyboard can take up a significant amount of screen space, and the app's layout is not automatically adjusted to accommodate it. As a result, the `TextField` widget can become hidden behind the soft keyboard, making it difficult or impossible for the user to interact with it.

User Registration

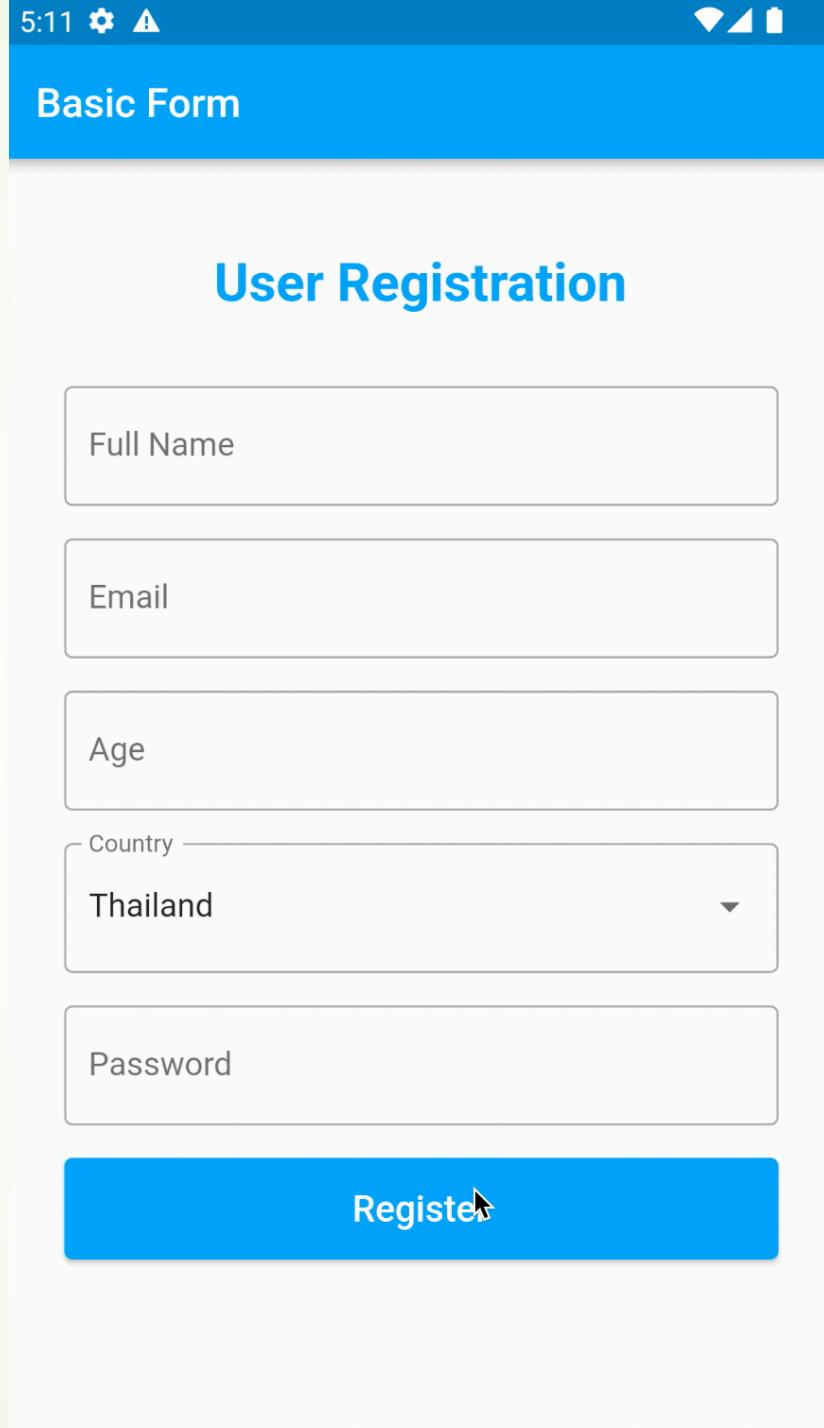
The screenshot shows a mobile application interface titled "User Registration". The form consists of several input fields: "Full Name" (placeholder: Enter your name), "Email" (placeholder: Email), "Age" (placeholder: Age), "Country" (dropdown menu showing "Thailand"), and "Password" (text field). A yellow and black diagonal striped overlay covers the bottom portion of the screen, with the text "BOTTOM OVERFLOWED BY 102 PIXELS" visible. At the bottom, a standard iOS-style keyboard is shown with keys for letters, numbers, punctuation, and a "done" button.

Soft Keyboard

- One common approach to solve the issue is to wrap the `TextField` widget with a `SingleChildScrollView`. This allows the user to scroll the content when the soft keyboard is displayed and prevents the `TextField` from being hidden.

`Scaffold(`

```
  appBar: AppBar(title: const Text('Basic Form')),  
  body: SingleChildScrollView(...)  
)
```



Working with Form - Conclusion

- ▶ **Managing form state:** Flutter's Form widget manages the state of the form for you. However, you need to use a GlobalKey<FormState> to access the state of the Form widget from the outside. It's important to manage the state of the form correctly, especially when you have multiple form fields.
- ▶ **Validating user input:** You can validate user input in a form by using the validator callback in TextFormField or DropdownButtonFormField. The validator function takes the input value and returns an error message if the input is invalid. You should use validation to ensure that the input values are of the expected type and format.
- ▶ **Submitting the form:** When the user submits the form, you need to validate the input, save the form data, and process it. You can do this by calling the validate() method of the form's GlobalKey<FormState> to check for errors, and then calling the onSaved callback to save the form data.
- ▶ **Providing feedback:** It's important to provide feedback to the user when they submit a form. You can use a Snackbar to show a message indicating whether the form submission was successful or not.

Working with Form - Conclusion

- ▶ **Managing keyboard focus:** When the user is filling out a form, the keyboard may be visible, covering part of the screen. You should ensure that the form fields are positioned correctly so that the user can see what they are typing. You can use the FocusNode and FocusScope widgets to manage keyboard focus and visibility.
- ▶ **Accessibility:** Make sure your form is accessible to all users, including those who rely on assistive technology. For example, you can add labels to form fields to help screen readers, and make sure that your forms can be navigated using only the keyboard.
- ▶ **Styling:** You can style your form to match the design of your app, using the decoration property of TextFormField or DropdownButtonFormField. It's important to use colors and fonts that are easy to read, and to ensure that your form looks good on a variety of devices and screen sizes.