


# MOBILE PROGRAMMING WITH FLUTTER - DAY4

เริ่มพัฒนาแอปพลิเคชันบนมือถือด้วย Flutter

## เกณฑ์คะแนน

ส่วนประกอบ (Component)	สัดส่วน (%)	วิธีการวัดผล
<b>1. Class Engagement / Micro-Challenges &amp; Home work</b>	<b>20%</b>	ถาม-ตอบในคลาส / Show & Tell งานในคาบ / เช็คงานท้ายคาบ (Pass/Fail) / การบ้าน
<b>2. Term Project</b> (Group Work)	<b>40%</b>	Proposal / Progress Check / Final Demo & Pitching
<b>3. Midterm Exam</b> (Practical Lab)	<b>20%</b>	โจทย์สร้าง UI/Logic/คุณภาพ Code ในห้องปฏิบัติการคอมพิวเตอร์
<b>4. Final Exam</b> (Scenario/Case Study)	<b>20%</b>	โจทย์ออกแบบ Architecture / แก้ Bug / เขียน Flow (ไม่เน้น Code ยาว)

```
android_context_gl_impeller.cc(104)] Using
```

 Dart DevTools includes additional tools for debugging and profiling Flutter apps, including a Widget Inspector. Try it?



Source: Dart

Open

Always Open

Not Now

Never Ask

Open

Go Live

Pixel 6 API 32 (android-arm64 emulator)



Go Live



Prettier



# สร้างฟอร์ม และ กำหนดสไตล์

[Cookbook](#) > [Forms](#) > Create and style a text field



# Create and style a text field

How to implement a text field.

Text fields allow users to type text into an app. They are used to build forms, send messages, create search experiences, and more. In this recipe, explore how to create and style text fields.

Flutter provides two text fields: `TextField` and `TextFormField`.

# TextField

`TextField` is the most commonly used text input widget.

By default, a `TextField` is decorated with an underline. You can add a label, icon, inline hint text, and error text by supplying an `InputDecoration` as the `decoration` property of the `TextField`. To remove the decoration entirely (including the underline and the space reserved for the label), set the `decoration` to null.

```
TextField(  
  decoration: InputDecoration(  
    border: OutlineInputBorder(),  
    hintText: 'Enter a search term',  
  ),  
),
```

dart

To retrieve the value when it changes, see the [Handle changes to a text field](#) recipe.

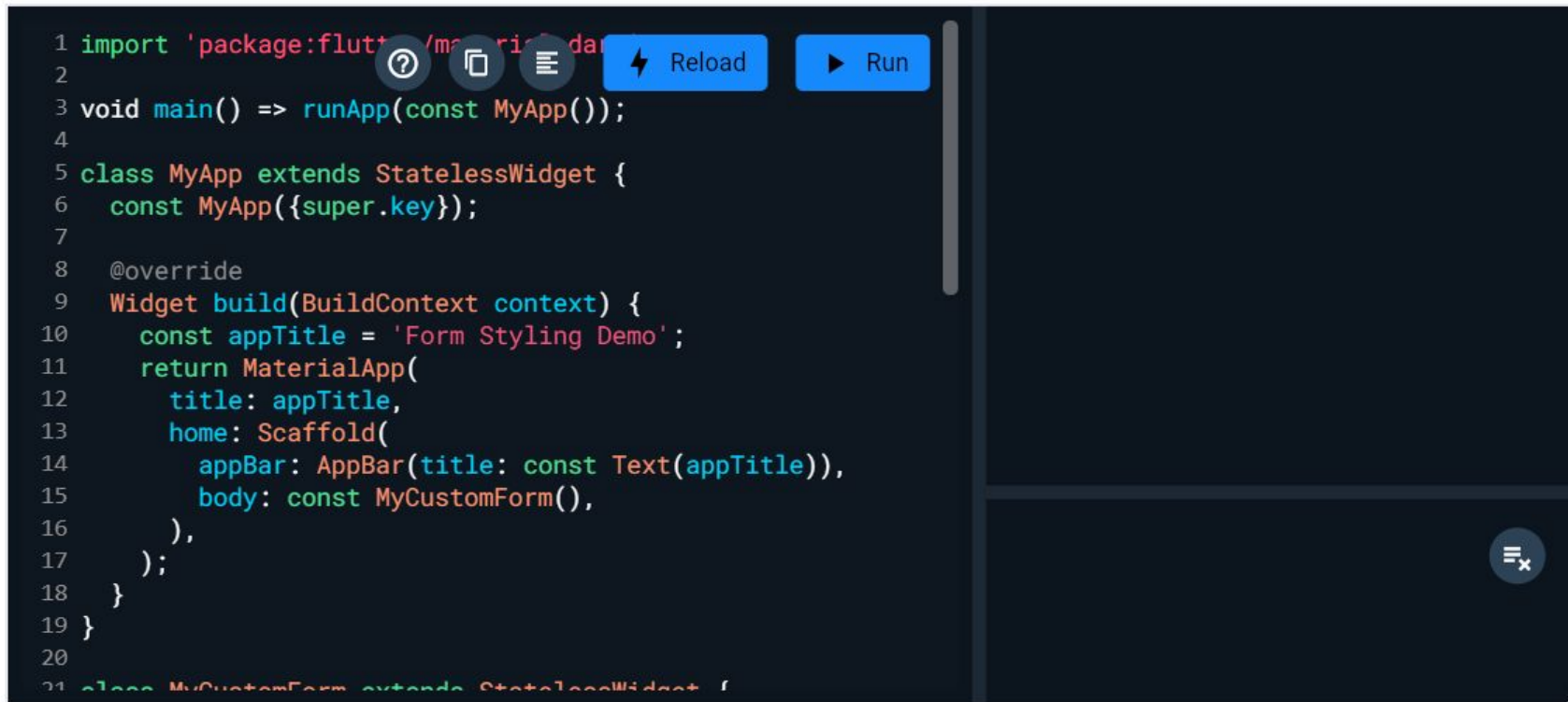
# TextFormField

`TextFormField` wraps a `TextField` and integrates it with the enclosing `Form`. This provides additional functionality, such as validation and integration with other `FormField` widgets.

```
TextFormField(  
  decoration: const InputDecoration(  
    border: UnderlineInputBorder(),  
    labelText: 'Enter your username',  
  ),  
),
```



# Interactive example



```

1 import 'package:flutter/material.dart';
2
3 void main() => runApp(const MyApp());
4
5 class MyApp extends StatelessWidget {
6   const MyApp({super.key});
7
8   @override
9   Widget build(BuildContext context) {
10     const appTitle = 'Form Styling Demo';
11     return MaterialApp(
12       title: appTitle,
13       home: Scaffold(
14         appBar: AppBar(title: const Text(appTitle)),
15         body: const MyCustomForm(),
16       ),
17     );
18   }
19 }
20
21 class MyCustomForm extends StatelessWidget {

```

For more information on input validation, see the [Building a form with validation](#) recipe.

แล้วเราจะดึง Value ออกมาอย่างไร ?

[Cookbook](#) > [Forms](#) > Retrieve the value of a text field



# Retrieve the value of a text field

How to retrieve text from a text field.

In this recipe, learn how to retrieve the text a user has entered into a text field using the following steps:

1. Create a `TextEditingController`.
2. Supply the `TextEditingController` to a `TextField`.
3. Display the current value of the text field.

## 1. Create a `TextEditingController`

To retrieve the text a user has entered into a text field, create a `TextEditingController` and supply it to a `TextField` or `TextFormField`.

### Important

Call `dispose` of the `TextEditingController` when you've finished using it. This ensures that you discard any resources used by the object.

```
// Define a custom Form widget.  
class MyCustomForm extends StatefulWidget {  
  const MyCustomForm({super.key});  
  
  @override  
  State<MyCustomForm> createState() => _MyCustomFormState();  
}
```

```
// Define a corresponding State class.  
// This class holds the data related to the Form.  
class _MyCustomFormState extends State<MyCustomForm> {  
  // Create a text controller and use it to retrieve the current value  
  // of the TextField.  
  final myController = TextEditingController();  
  
  @override  
  void dispose() {  
    // Clean up the controller when the widget is disposed.  
    myController.dispose();  
    super.dispose();  
  }  
  
  @override  
  Widget build(BuildContext context) {  
    // Fill this out in the next step.  
  }  
}
```

## 2. Supply the `TextEditingController` to a `TextField`

Now that you have a `TextEditingController`, wire it up to a text field using the `controller` property:

```
return TextField(controller: myController);
```



### 3. Display the current value of the text field

After supplying the `TextEditingController` to the text field, begin reading values. Use the `text` property provided by the `TextEditingController` to retrieve the `String` that the user has entered into the text field.

The following code displays an alert dialog with the current value of the text field when the user taps a floating action button.

```

FloatingActionButton(
  // When the user presses the button, show an alert dialog containing
  // the text that the user has entered into the text field.
  onPressed: () {
    showDialog(
      context: context,
      builder: (context) {
        return AlertDialog(
          // Retrieve the text that the user has entered by using the
          // TextEditingController.
          content: Text(myController.text),
        );
      },
    );
  },
  tooltip: 'Show me the value!',
  child: const Icon(Icons.text_fields),
),

```

# Challenge A: ปุ่มล้างข้อความ (Clear)

# จัดการกับ Event ภายใน Text Field

[Cookbook](#) > [Forms](#) > Handle changes to a text field



# Handle changes to a text field

How to detect 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. For example, you might want to build a search screen with autocomplete functionality where you want to update the results as the user types.

How do you run a callback function every time the text changes? With Flutter, you have two options:

1. Supply an `onChanged()` callback to a `TextField` or a `TextFormField`.
2. Use a `TextEditingController`.

# 1. Supply an `onChanged()` callback to a `TextField` or a `TextFormField`

The simplest approach is to supply an `onChanged()` callback to a `TextField` or a `TextFormField`. Whenever the text changes, the callback is invoked.

In this example, print the current value and length of the text field to the console every time the text changes.

It's important to use `characters` when dealing with user input, as text may contain complex characters. This ensures that every character is counted correctly as they appear to the user.

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

dart

## 2. Use a `TextEditingController`

A more powerful, but more elaborate approach, is to supply a `TextEditingController` as the `controller` property of the `TextField` or a `TextFormField`.

To be notified when the text changes, listen to the controller using the `addListener()` method using the following steps:

1. Create a `TextEditingController`.
2. Connect the `TextEditingController` to a text field.
3. Create a function to print the latest value.
4. Listen to the controller for changes.

```
// Define a custom Form widget.
class MyCustomForm extends StatefulWidget {
  const MyCustomForm({super.key});

  @override
  State<MyCustomForm> createState() => _MyCustomFormState();
}

// Define a corresponding State class.
// This class holds data related to the Form.
class _MyCustomFormState extends State<MyCustomForm> {
  // Create a text controller. Later, use it to retrieve the
  // current value of the TextField.
  final myController = TextEditingController();

  @override
  void dispose() {
    // Clean up the controller when the widget is removed from the
    // widget tree.
    myController.dispose();
    super.dispose();
  }

  @override
  Widget build(BuildContext context) {
    // Fill this out in the next step.
  }
}
```

```
// Define a custom Form widget.  
class MyCustomForm extends StatefulWidget {  
  const MyCustomForm({super.key});  
  
  @override  
  State<MyCustomForm> createState() => _MyCustomFormState();  
}
```

```
// Define a corresponding State class.  
// This class holds data related to the Form.  
class _MyCustomFormState extends State<MyCustomForm> {  
  // Create a text controller. Later, use it to retrieve the  
  // current value of the TextField.  
  final myController = TextEditingController();  
  
  @override  
  void dispose() {  
    // Clean up the controller when the widget is removed from the  
    // widget tree.  
    myController.dispose();  
    super.dispose();  
  }  
  
  @override  
  Widget build(BuildContext context) {  
    // Fill this out in the next step.  
  }  
}
```

### Note

Remember to dispose of the `TextEditingController` when it's no longer needed. This ensures that you discard any resources used by the object.

## Connect the `TextEditingController` to a text field

Supply the `TextEditingController` to either a `TextField` or a `TextFormField`. Once you wire these two classes together, you can begin listening for changes to the text field.

```
TextField(controller: myController),
```

dart

### Note

Remember to dispose of the `TextEditingController` when it's no longer needed. This ensures that you discard any resources used by the object.

## Connect the `TextEditingController` to a text field

Supply the `TextEditingController` to either a `TextField` or a `TextFormField`. Once you wire these two classes together, you can begin listening for changes to the text field.

```
TextField(controller: myController),
```

dart

## Create a function to print the latest value

You need a function to run every time the text changes. Create a method in the `_MyCustomFormState` class that prints out the current value of the text field.

```
void _printLatestValue() {  
  final text = myController.text;  
  print('Second text field: $text (${text.characters.length})');  
}
```

dart

## Listen to the controller for changes

Finally, listen to the `TextEditingController` and call the `_printLatestValue()` method when the text changes. Use the `addListener()` method for this purpose.

Begin listening for changes when the `_MyCustomFormState` class is initialized, and stop listening when the `_MyCustomFormState` is disposed.

```
@override
void initState() {
    super.initState();

    // Start listening to changes.
    myController.addListener(_printLatestValue);
}
```

```
@override
void dispose() {
    // Clean up the controller when the widget is removed from the widget tree.
    // This also removes the _printLatestValue listener.
    myController.dispose();
    super.dispose();
}
```

## Challenge B: โหมด “พิมพ์ปั๊บโชว์ปั๊บ”

เราต้องรู้จัก Focus

[Cookbook](#) > [Forms](#) > Focus and text fields

# Focus and text fields

How focus works with text fields.

When a text field is selected and accepting input, it is said to have "focus." Generally, users shift focus to a text field by tapping, and developers shift focus to a text field programmatically by using the tools described in this recipe.

Managing focus is a fundamental tool for creating forms with an intuitive flow. For example, say you have a search screen with a text field. When the user navigates to the search screen, you can set the focus to the text field for the search term. This allows the user to start typing as soon as the screen is visible, without needing to manually tap the text field.

In this recipe, learn how to give the focus to a text field as soon as it's visible, as well as how to give focus to a text field when a button is tapped.

# Focus a text field as soon as it's visible

To give focus to a text field as soon as it's visible, use the `autofocus` property.

```
TextField(  
  autofocus: true,  
);
```

For more information on handling input and creating text fields, see the [Forms](#) section of the cookbook.

# Focus a text field when a button is tapped

Rather than immediately shifting focus to a specific text field, you might need to give focus to a text field at a later point in time. In the real world, you might also need to give focus to a specific text field in response to an API call or a validation error. In this example, give focus to a text field after the user presses a button using the following steps:

1. Create a `FocusNode`.
2. Pass the `FocusNode` to a `TextField`.
3. Give focus to the `TextField` when a button is tapped.

## 1. Create a `FocusNode`

First, create a `FocusNode`. Use the `FocusNode` to identify a specific `TextField` in Flutter's "focus tree." This allows you to give focus to the `TextField` in the next steps.

Since focus nodes are long-lived objects, manage the lifecycle using a `State` object. Use the following instructions to create a `FocusNode` instance inside the `initState()` method of a `State` class, and clean it up in the `dispose()` method:

```
// Define a custom Form widget.  
class MyCustomForm extends StatefulWidget {  
  const MyCustomForm({super.key});  
  
  @override  
  State<MyCustomForm> createState() => _MyCustomFormState();  
}
```

```
// Define a corresponding State class.
// This class holds data related to the form.
class _MyCustomFormState extends State<MyCustomForm> {
  // Define the focus node. To manage the lifecycle, create the FocusNode in
  // the initState method, and clean it up in the dispose method.
  late FocusNode myFocusNode;

  @override
  void initState() {
    super.initState();

    myFocusNode = FocusNode();
  }

  @override
  void dispose() {
    // Clean up the focus node when the Form is disposed.
    myFocusNode.dispose();

    super.dispose();
  }

  @override
  Widget build(BuildContext context) {
    // Fill this out in the next step.
  }
}
```

## 2. Pass the `FocusNode` to a `TextField`


Now that you have a `FocusNode`, pass it to a specific `TextField` in the `build()` method.

```
@override  
Widget build(BuildContext context) {  
  return TextField(focusNode: myFocusNode);  
}
```

### 3. Give focus to the `TextField` when a button is tapped

Finally, focus the text field when the user taps a floating action button. Use the `requestFocus()` method to perform this task.

```
FloatingActionButton(  
  // When the button is pressed,  
  // give focus to the text field using myFocusNode.  
  onPressed: () => myFocusNode.requestFocus(),  
),
```

1. ห้ามสร้าง FocusNode ใน `build()`  
 สร้างใน `initState()`
2. อย่าลืม `dispose()`



## The 5 dimensions of interaction design

The 5 dimensions of interaction design<sup>(1)</sup> is a useful model to understand what interaction design involves. Gillian Crampton Smith, an interaction design academic, first introduced the concept of four dimensions of an interaction design language, to which Kevin Silver, senior interaction designer at IDEXX Laboratories, added the fifth.

### 1D: Words

Words—especially those used in interactions, like button labels—should be meaningful and simple to understand. They should communicate information to users, but not too much information to overwhelm the user.

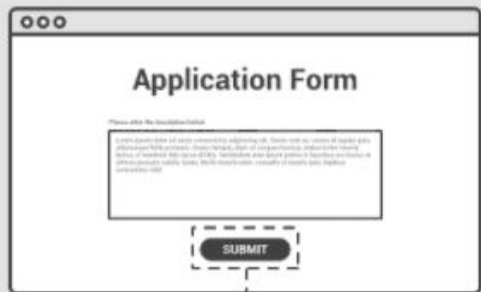
### 2D: Visual representations

This concerns graphical elements like images, [typography](#) and icons that users interact with. These usually supplement the words used to communicate information to users.





# 5 DIMENSIONS OF INTERACTION DESIGN



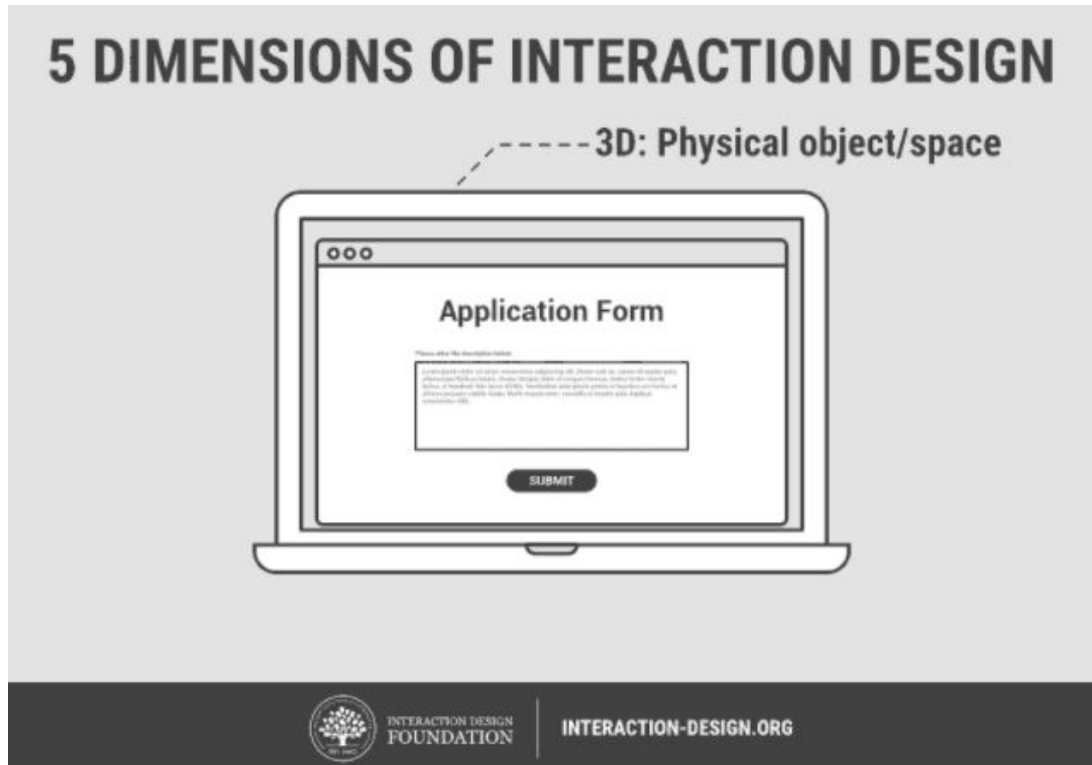
2D: Visual representation



INTERACTION DESIGN  
FOUNDATION

INTERACTION-DESIGN.ORG

**2D: Visual representations** รูปภาพ การออกแบบตัวอักษร  
ใช้เพื่อการสื่อความหมายเพิ่มเติม (เช่น ไอคอนต่าง ๆ)



**3D: Physical objects or space** ดูว่าผู้ใช้งานใช้ผลิตภัณฑ์ของเราผ่านอุปกรณ์ หรือ สถานที่ใดบ้าง ?

## 5 DIMENSIONS OF INTERACTION DESIGN

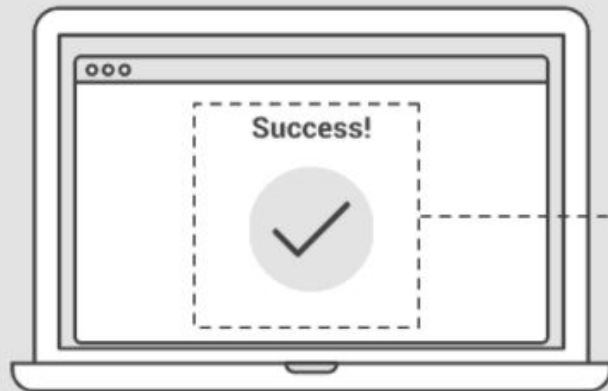


INTERACTION DESIGN  
FOUNDATION

INTERACTION-DESIGN.ORG

**4D: Time พุดถึงเรื่องเวลา** เวลาที่ผู้ใช้งานจะใช้แอปของเรา และผู้ใช้งานรู้ใหม่ว่ามันคืบหน้าไปเท่าไรแล้ว จนถึง เวลาของเทรนต์

## 5 DIMENSIONS OF INTERACTION DESIGN



5D: Behaviour  
(reaction)



INTERACTION DESIGN  
FOUNDATION

INTERACTION-DESIGN.ORG

**5D: Behaviour** พฤติกรรม และ การใช้งานผลิตภัณฑ์ของเรา  
จนถึง การตอบสนองทางอารมณ์หรือคำติชม

ตรวจสอบข้อความได้อย่างไร ?

# Build a form with validation

How to build a form that validates input.

Apps often require users to enter information into a text field. For example, you might require users to log in with an email address and password combination.

To make apps secure and easy to use, check whether the information the user has provided is valid. If the user has correctly filled out the form, process the information. If the user submits incorrect information, display a friendly error message letting them know what went wrong.

In this example, learn how to add validation to a form that has a single text field using the following steps:

1. Create a `Form` with a `GlobalKey`.
2. Add a `TextFormField` with validation logic.
3. Create a button to validate and submit the form.

# 1. Create a Form with a GlobalKey

Create a `Form`. The `Form` widget acts as a container for grouping and validating multiple form fields.

When creating the form, provide a `GlobalKey`. This assigns a unique identifier to your `Form`. It also allows you to validate the form later.

Create the form as a `StatefulWidget`. This allows you to create a unique `GlobalKey<FormState>()` once. You can then store it as a variable and access it at different points.

If you made this a `StatelessWidget`, you'd need to store this key *somewhere*. As it is resource expensive, you wouldn't want to generate a new `GlobalKey` each time you run the `build` method.

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

// Define a custom Form widget.
class MyCustomForm extends StatefulWidget {
  const MyCustomForm({super.key});

  @override
  MyCustomFormState createState() {
    return MyCustomFormState();
  }
}
```

```
// Define a corresponding State class.
// This class holds data related to the form.
class MyCustomFormState extends State<MyCustomForm> {
  // Create a global key that uniquely identifies the Form widget
  // and allows validation of the form.
  //
  // Note: This is a `GlobalKey<FormState>`,
  // not a GlobalKey<MyCustomFormState>.
  final _formKey = GlobalKey<FormState>();

  @override
  Widget build(BuildContext context) {
    // Build a Form widget using the _formKey created above.
    return Form(
      key: _formKey,
      child: const Column(
        children: <Widget>[
          // Add TextFormField and ElevatedButton here.
        ],
      ),
    );
  }
}
```

 Tip

Using a `GlobalKey` is the recommended way to access a form. However, if you have a more complex widget tree, you can use the `Form.of()` method to access the form within nested widgets.

## 2. Add a `TextFormField` with validation logic

Although the `Form` is in place, it doesn't have a way for users to enter text. That's the job of a `TextFormField`. The `TextFormField` widget renders a material design text field and can display validation errors when they occur.

Validate the input by providing a `validator()` function to the `TextFormField`. If the user's input isn't valid, the `validator` function returns a `String` containing an error message. If there are no errors, the validator must return null.

For this example, create a `validator` that ensures the `TextFormField` isn't empty. If it is empty, return a friendly error message.



```
TextField(  
  // The validator receives the text that the user has entered.  
  validator: (value) {  
    if (value == null || value.isEmpty) {  
      return 'Please enter some text';  
    }  
    return null;  
  },  
)
```

### 3. Create a button to validate and submit the form

Now that you have a form with a text field, provide a button that the user can tap to submit the information.

When the user attempts to submit the form, check if the form is valid. If it is, display a success message. If it isn't (the text field has no content) display the error message.



```
ElevatedButton(  
  onPressed: () {  
    // Validate returns true if the form is valid, or false otherwise.  
    if (_formKey.currentState!.validate()) {  
      // If the form is valid, display a snackbar. In the real world,  
      // you'd often call a server or save the information in a database.  
      ScaffoldMessenger.of(context).showSnackBar(  
        const SnackBar(content: Text('Processing Data')),  
      );  
    }  
  },  
  child: const Text('Submit'),  
)
```

## How does this work?

To validate the form, use the `_formKey` created in step 1. You can use the `_formKey.currentState` accessor to access the `FormState`, which is automatically created by Flutter when building a `Form`.

The `FormState` class contains the `validate()` method. When the `validate()` method is called, it runs the `validator()` function for each text field in the form. If everything looks good, the `validate()` method returns `true`. If any text field contains errors, the `validate()` method rebuilds the form to display any error messages and returns `false`.

# การบ้านสำหรับเช็คชื่อรอบถัดไป ..

สร้างหน้าฟอร์ม 1 หน้า ให้ได้เหมือนกับแอปต้นฉบับที่  
หามา พร้อมทั้งสามารถดึงข้อมูลที่ User กรอกมา  
แสดงผลได้ในฟอร์มนั้น ๆ หรือ แสดงผลในหน้าถัดไป

PROJECT รวมกลุ่ม 3 คน

Technical 25% Idea 15%