salesians
SARRIÀ

Programación multimedia y dispositivos móviles.

Flutter: Forms

Alumno: Damian Altamirano Ontiveros Rivero.

Índice

0. Introducción

En esta ocasión, veremos la lógica detrás de la creación de formulario en flutter y veremos como podemos podemos aprovechar los diferentes recursos que y librerías que nos ofrece el mismo framework para la creación de formularios interactivos y escalables.

1. Librerías Principales

Antes de empezar con el desarrollo de nuestros proyectos, empezaremos hablando de librerías imprescindibles para la construcción de nuestros formularios con widgets de una manera mas simple y escalable.

1.1 flutter_form_builder

Es una herramienta fundamental en nuestros proyectos Flutter. Desarrollado por los creadores del framework, esta librería ofrece una amplia gama de widgets preconfigurados que simplifican la creación de formularios interactivos, validados y altamente personalizables. Su objetivo principal es acelerar el desarrollo manteniendo la coherencia visual y funcional en toda la aplicación.

Entre sus componentes más destacados se encuentran:

- **FormBuilderCheckbox**: Campo de selección individual tipo checkbox.
- **FormBuilderCheckboxGroup**: Grupo de checkboxes para selección múltiple.
- **FormBuilderChoiceChip**: Chips que funcionan como botones de opción (radio).
- **FormBuilderDateRangePicker**: Selector de rangos de fechas.
- **FormBuilderDateTimePicker**: Entrada combinada de fecha y hora.
- **FormBuilderDropdown**: Menú desplegable para seleccionar un valor de una lista.
- **FormBuilderFilterChip**: Chips que actúan como checkboxes.
- **FormBuilderRadioGroup**: Grupo de botones de opción para selección única.
- **FormBuilderRangeSlider**: Selector de rango numérico mediante slider.
- **FormBuilderSlider**: Slider para seleccionar un valor numérico.
- **FormBuilderSwitch**: Interruptor tipo On/Off.
- **FormBuilderTextField**: Campo de texto con estilo Material Design.

Estos widgets están diseñados para integrarse fácilmente con el sistema de validación y estado de los formularios, permitiendo una experiencia de desarrollo más fluida y escalable.

1.2 form_builder_extra_fields

Este amplía las capacidades del paquete base **flutter_form_builder**, ofreciendo widgets especializados que cubren casos de uso más complejos y específicos. Estos componentes permiten integrar funcionalidades avanzadas sin necesidad de desarrollarlas desde cero, lo que acelera el desarrollo y mejora la experiencia del usuario final.

Entre los widgets más útiles se encuentran:

- **FormBuilderColorPicker**: Selector de color con soporte para paletas, RGB y HSV.
- **FormBuilderCupertinoDateTimePicker**: Selector de fecha y hora con estilo iOS.
- **FormBuilderImagePicker**: Permite seleccionar imágenes desde la galería o la cámara.
- **FormBuilderPhoneField**: Campo de entrada para números de teléfono con validación internacional.
- **FormBuilderRating**: Componente para capturar valoraciones mediante estrellas u otros íconos.
- **FormBuilderSignaturePad**: Área de dibujo para capturar firmas digitales.
- **FormBuilderTouchSpin**: Selector numérico con botones de incremento/decremento.
- **FormBuilderTypeAhead**: Campo de texto con sugerencias automáticas mientras el usuario escribe.

Estos widgets están diseñados para integrarse perfectamente con el sistema de validación y estado de flutter_form_builder, manteniendo la coherencia y escalabilidad del formulario.

2. Creación de Formularios

En este apartado veremos como nuestros como crear y diseñar proyectos con formularios interactivos usando los paquetes ***flutter_form_builder*** y ***forms_builder_extra_fields.***

2.1 Formulario 1: Cuadros de Dialogo

2.1.1 Estructura inicial

En este formulario, veremos los diferentes cuadros de dialogo que existen y sus características, introduciendo un TextField del cual tomaremos los valores introducidos en el para mostrarlos en los diferentes cuadros de dialogo.

Primero, importamos esta librería base que nos será útil para construir nuestro formulario:

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

Ahora, es importante empezar con la estructura principal del programa. Empezamos declarando la expresión “runApp()” que se inicializar en cuando se desencadene el main().

“runApp()” es una función que proporciona Flutter para inicializar la aplicación y montar el widget que se le pasa como argumento, que en este caso será “const MyApp()”.

La clase “MyApp” es una subclase de “StatelessWidget”, una clase predefinida de Flutter para la construcción de aplicaciones, junto con “StatefulWidegt”. Usaremos “StatelessWidget” ya que una de sus características principales es que tiene un ciclo de vida muy simple y sirve para aplicativos con un estado interno inmutable.

Dentro de la clase sobre escribiremos el metodo “build” el cual nos retornará un “Widget”. Como argumento de este, definiremos el “BuildContext”. Flutter llamará a este método cada vez que necesitemos renderizar el widget (en el caso de “StatefulWidget”).

“BuildContext” es un objeto que representa la ubicación de un widget dentro del “widgets tree”. En otras palabras, es como si fuera una referencia única de nuestro widget el cual nos será útil para poder acceder a este dentro del “widgets tree” y a la información de temas, localización y navegación.

Por último, retornaremos una constante que será un widget de nivel superior, que configurará nuestra aplicación con el estilo y comportamiento del Material Designe (Google style).

En nuestro caso, definiremos el titulo y el widget raíz de nuestra aplicación al iniciarla, que en nuestro caso será “MyCustomForm()”.

```
Run | Debug | Profile
void main() => runApp(const MyApp());

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      title: 'Retrieve a value in a text field',
      home: MyCustomForm(),
    ); // MaterialApp
  }
}
```

Ahora, iremos con la configuración de “MyCustomForm”, el cual heredará de la clase “StatefulWidget”.

Dentro de esta clase, sobre escribiremos el método “createState()”, el cual nos retornará un “State<T>” el cual representa el estado mutable de nuestro “StatefulWidget”, qué en este caso, significa que el estado pertenece a “MyCustomForm”. El widget de por si será inmutable, pero lo que irán cambiando serán sus estados. Donde vivirán y esos estados mutables será dentro de “_MyCustomFormState()”, el cual extiende nuestra case “State<MyCustomForm>” por lo que estarán vinculados.

```
class MyCustomForm extends StatefulWidget {
  const MyCustomForm({super.key});
  @override
  State<MyCustomForm> createState() => _MyCustomFormState();
}
```

Una vez hecho todo esto, nos tocará configurar la clase “_MyCustomFormState()” donde empezaremos a alojar la estructura de widgets y sus funcionalidades.

Primero, heredaremos de “State<MyCustomForm>”. Luego definiremos el objeto “TextEditingController()” como “final” para todo nuestro formulario. Este objeto es el encargado de gestionar el texto dentro de un “TextField” o un “TextFormField”. Actúa como puente entre el estado visual del widget y el estado del widget, por lo que si deseamos obtener el valor de estos widgets solo tendremos que acceder a la propiedad “TextEditingController.text”.

además de esto, sobre escribiremos el método “dispose()”, el cual es parte del ciclo de vida del “State” y se activa cuando el widget de estado se elimina completamente del “widgets tree”. Es importante y obligatorio gestionar la memoria y los recursos de nuestros widgets manualmente ya que, si no lo hiciéramos, podríamos crear fugas de memoria provocando que tengamos objetos no visibles aún en memoria.

```
class _MyCustomFormState extends State<MyCustomForm> {
  final myController = TextEditingController();

  @override
  void dispose() {
    myController.dispose();
    super.dispose();
  }
```

Seguiremos dentro de nuestra clase "_MyCustomFormState", pero ahora sobre escribiendo el metodo build, el cual describirá la interfaz de usuario de nuestra app.

Definimos el BuildContext del metodo retornaremos un "Scaffold" (Andamio), el cual nos servira para encapsular otros widgets dentro de este.

Inicialmente definiremos el appBar el body de nuestra app.

Dentro del body definiremos uno de los "hijos" que tendra que será un "TextField", el cual tendra como controlador nuestra final "TextEditingController()"

```
@override
Widget build(BuildContext context) {
  return Scaffold(
      appBar: AppBar(
        title: const Text('Recuperar el valor d\'un camp de text'),
      ), // AppBar
      body: Padding(
        padding: const EdgeInsets.all(16.0),
        child: TextField(
          controller: myController,
        ), // TextField
      ), // Padding
```

Siguiendo con la configuración del Scaffold, definiremos la propiedad "FloationgActionButton" como una "Row", la cual contendrá una array de "childrens" que será "FloatingActionButton".

```
floatingActionButton:
    Row(mainAxisAlignment: MainAxisAlignment.center, children: [
  FloatingActionButton(
```

Ya con todo esto, podremos empezar con la contrucción de nuestros botones. Mencionar el objeto "FloatingActionButton" tendrá tres propiedades fundamentales:

- onPressed. - Evento del botón que se desencadena cuando presionamos sobre este.
- Tooltip. - Este mostrará un pequeño mensaje que servirá como guía para el usuario cuando este sobre el botón.

- child. – A este le asignaremos el icono que deseamos ver dentro de él.

2.1.2 Simple Dialog

Este botón mostrará un “SimpleDialog”, él cual no es más que un cuadro de dialogo simple que solo mostrará el mensaje que decidamos definirle.

```
FloatingActionButton(
  onPressed: () {
    showDialog(
      context: context,
      barrierDismissible: true,
      builder: (context) {
        return SimpleDialog(
            title: const Text("Simple Dialog"),
            shape: RoundedRectangleBorder(
                borderRadius: BorderRadius.circular(10.0)), // Round
            children: [
              Padding(
                padding:
                    EdgeInsetsGeometry.symmetric(horizontal: 24.0),
                child: Text(myController.text),
              ), // Padding
            ]); // SimpleDialog
      },
    );
  },
  tooltip: 'Show Simple Dialog',
  child: const Icon(Icons.text_fields),
), // FloatingActionButton
```

Dentro del evento onPressed() definiremos la función “showDialog” el cual deberá de retornarnos un objeto “SimpleDialog”, además de definirle nuestro context dentro del “builder”.

Dentro del objeto “SimpleDialog”, definiremos el “title”, la forma de este (“shape”) y el contenido (“children”).

```
const SizedBox(width: 16),
```

Como punto clave, entre botón y botón, dejaremos un espacio para que visualmente no este empiñados.

2.1.3 Alert Dialog

Un cuadro de dialogo parecido al “Simple Dialog”, pero con la peculiaridad de que estos suelen mostrarse como una alerta crítica dentro del flujo de la aplicación para el usuario, restringiendo la salida de este solo con el botón dentro del dialogo.

```
FloatingActionButton(
  onPressed: () {
    showDialog(
        context: context,
        barrierDismissible: false,
        builder: (context) {
          return AlertDialog(
            title: const Text("Alert Dialog"),
            shape: RoundedRectangleBorder(
                borderRadius: BorderRadius.circular(10.0)),
            content: Text(myController.text),
            actions: [
              ElevatedButton(
                onPressed: () {
                  Navigator.of(context).pop();
                },
                child: const Text("OK"),
              ) // ElevatedButton
            ],
          ); // AlertDialog
        });
  },
  tooltip: 'Show Alert Dialog',
  child: const Icon(Icons.text_fields),
), // FloatingActionButton
```

Al igual que antes, definiremos las propiedades estandares del “FloatingActionButton”, solo que esta vez retornaremos un “AlertDialog” cuando se desencadene el evento “showDialog()”. Dentro de este definiremos el contenido, el cual sera el que esté dentro de “MyController.text”, pero agregaremos un “ElevatedButton”, el cual nos permitirá salir del cuadro de diálogo.

2.1.4 Snack Bar Message

Un tipo de mensaje que aparecerá como una barra en la parte inferior de la aplicación durante unos segundos, útil para mensajes rapidos y sin comprometer el flujo de la app.

```
FloatingActionButton(
  onPressed: () {
    ScaffoldMessenger.of(context).showSnackBar(SnackBar(
      content: Text(myController.text),
      duration: Duration(seconds: 2),
    )); // SnackBar
  },
  tooltip: 'Show Snack Bar',
  child: const Icon(Icons.text_fields),
), // FloatingActionButton
```

Definimos las propiedades estandars y dentro de la propiedad “onPressed” instanciaremos el objeto “ScaffoldMessenger” el cual tendra un metódo llamado ”showSnackBar”, el cuál servíra como remplazo del “showDialog()”.

A este metodo le pasaremos como argumento un “SnackBar”, el cual tendra como contenido el valor del “TextField” y una duración de 2 segundos.

2.1.5 Modal Button Sheet

A diferencias de los demás cuadros diálogos, este aparecerá como un panel emergente desde la parte inferior, el cual solo podrá ser cerrado por un botón.

```
FloatingActionButton(
  onPressed: () {
    showModalBottomSheet(
        backgroundColor: Colors.transparent,
        context: context,
        builder: (BuildContext context) {
          return Container(
              decoration: const BoxDecoration(
                color: Colors.amber,
                borderRadius: BorderRadius.only(
                  topLeft: Radius.circular(25),
                  topRight: Radius.circular(25),
                ), // BorderRadius.only
              ), // BoxDecoration
              height: 200,
              child: Center(
                child: Column(
                  mainAxisAlignment: MainAxisAlignment.center,
                  mainAxisSize: MainAxisSize.min,
                  children: <Widget>[
                    Text(myController.text),
                    ElevatedButton(
                      onPressed: () => Navigator.of(context).pop(),
                      child: const Text('Close'),
                    ) // ElevatedButton
                  ], // <Widget>[]
                ), // Column
              )); // Center // Container
        });
  },
  tooltip: 'Show the value!',
  child: const Icon(Icons.text_fields),
), // FloatingActionButton
```

Aquí, definiremos estilos, pading, tamaños y los eventos del “ElevatedButton”

2.1.6 Demo

Con esto, ya habremos acabado nuestro formulario. Si deseas ver una demo de este lo encontraras en el apartado de **Recursos > Demo Actividad_01**

2.2 Formulario 2: Construcción de un formulario básico.

Para la construcción de un formulario, usaremos una librería que nos facilitará mucho el desarrollo y construcción de nuestro formulario, el cual será flutter_form_builder

2.2.1 Estructura inicial

Antes que nada, configuraremos el archivo “pubspec.yaml” para agregar la referencia a nuestra librería.

```
dependencies:
  flutter:
    sdk: flutter

  # The following adds the Cupertino Icons f
  # Use with the CupertinoIcons class for iO
  cupertino_icons: ^1.0.8
  flutter_form_builder: ^10.2.0
```

Luego ejecutaremos en la terminal el comando:

|| flutter pub get

el cual descargará los packages que estén declarados dentro del archivo pubspec.yaml.

Una vez hecho esto, solo faltará importar las librerías dentro de nuestro archivo main.dart.

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

Con este punto de partida, replicaremos la estructura de nuestro anterior programa haciendo algunas modificaciones extras.

```
Run | Debug | Profile
void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'M0489 - Apps - Form (A)',
      theme: ThemeData(
        colorScheme: ColorScheme.fromSeed(seedColor: Colors.blue),
        useMaterial3: true,
      ), // ThemeData
      home: MyHomePage(),
    ); // MaterialApp
  }
}
```

Partimos del método “main()” que nos retornará “runApp()”. Luego, definiremos la clase “MyApp” y esta heredará de “StatelessWidget”. Por lo demás, lo haremos como en la anterior estructura .

Ahora, configuraremos la clase “MyHomePage” que heredará de “StatelessWidget”, por lo que nuestro formulario no cambiará modularmente. Dentro de esta definiremos el titulo y un

GlobalKey<FormBuilderState>, la cual proviene del paquete flutter_form_builder y nos permite interactuar con el estado interno del FormBuilder remotamente.

Luego, sobre escribiremos la función build(), la cual tendra definido como argumento un "BuildContext" y esta retornará un "Scaffold". Añadiremos una appBar inicial con un titulo y tema estándar.

```
class MyHomePage extends StatelessWidget {
  MyHomePage({super.key});
  final String title = 'Salesians Sarrià 25/26';
  final _formKey = GlobalKey<FormBuilderState>();

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        backgroundColor: Theme.of(context).colorScheme.inversePrimary,
        title: Text(title),
      ), // AppBar
```

Dentro del "body", declararemos un "SingleChildScrollView", un objeto que permite scrollear una única columna o contenido cuando este exceda el tamaño visible.

Dentro de esta, instanciaremos una Columna la cual contendrá como "hijos" una lista de Widgets, de construiremos con ayuda del FormBuilder.

```
body: SingleChildScrollView(
  child: Column(
    crossAxisAlignment: CrossAxisAlignment.start,
    children: <Widget>[
      const FormTitle(),
      FormBuilder(
        key: _formKey,
        child: Padding(
          padding: const EdgeInsets.only(left: 20, right: 20),
          child: Column(
            crossAxisAlignment: CrossAxisAlignment.start,
            children: [
```

Una vez hecho esto, podremos empezar a construir los diferentes elementos de nuestro formulario.

2.2.2 Radio Group

Este es un objeto que contiene un conjunto de botones que sólo permiten seleccionar una sola opción de varias. Usado para cuando tenemos alternativas mutuamente excluyentes.

```
//------------------------------------------------
FormLabelGroup(
  title: 'Please provide the speed of vehicle?',
  subtitle: 'please select one option given below',
), // FormLabelGroup
FormBuilderRadioGroup(
  decoration:
      const InputDecoration(border: InputBorder.none),
  name: "speed",
  orientation: OptionsOrientation.vertical,
  // separator: const Padding(padding: EdgeInsets.all(20)),
  options: const [
    FormBuilderFieldOption(value: 'abvoe 40km/h'),
    FormBuilderFieldOption(value: 'below 40km/h'),
    FormBuilderFieldOption(value: '0km/h')
  ],
  onChanged: (String? value) {
    debugPrint(value);
  },
), // FormBuilderRadioGroup
```

Empezaremos creando un “FormLabelGroup”, el cual nos permitirá crear un bloque se texto personalizado. Luego instanciaremos la clase “FormBuilderRadioGroup” al cual podremos definir su aspecto (decoration), el nombre, la orientación, las opciones (las cuales tendrán que definirse como objetos “FormBuilderFieldOption”) y la acción que desencadenará el evento “onChanged”.

2.2.3 Text Field

Un objeto para almacenar campos de texto que pueden ser rellenados a los cuales es posible acceder a su valor.

```
//------------------------------------------------
FormLabelGroup(title: 'Enter remarks'),
FormBuilderTextField(
  name: 'remark',
  decoration: InputDecoration(
    hintText: 'Enter your remarks',
    border: OutlineInputBorder(
      borderRadius: BorderRadius.circular(10.0),
      borderSide: const BorderSide(
        width: 0,
        style: BorderStyle.none,
      ), // BorderSide
    ), // OutlineInputBorder
    filled: true,
  ), // InputDecoration
  onChanged: (String? value) {
    debugPrint(value);
  },
), // FormBuilderTextField
```

Para construir este objeto basta con llamar a “FormBuilderTextField”, donde podremos asignar el nombre, y el evento “onChanged”, el cual irá imprimiendo el valor que le vayamos ingresando.

2.2.4 Drop Down

Un Drop Down nos sirve para mostrar opciones en una lista desplegable donde solo se puede escoger una sola. Es ideal cuando queremos mostrar muchas opciones, pero queremos ahorrar espacio en la interfaz.

```
FormBuilderDropdown(
  name: 'highspeed',
  decoration: InputDecoration(
    hintText: 'Select option',
    border: OutlineInputBorder(
      borderRadius: BorderRadius.circular(10.0),
    ), // OutlineInputBorder
  ), // InputDecoration
  items: const [
    DropdownMenuItem(value: 'high', child: Text('High')),
    DropdownMenuItem(
        value: 'medium', child: Text('Medium')), // DropdownMe
    DropdownMenuItem(value: 'low', child: Text('Low')),
  ],
  onChanged: (String? value) {
    debugPrint(value);
  },
), // FormBuilderDropdown
```

Llamamos al método “FormBuilderDropDown” y definimos las propiedades del nombre, el estilo, los ítems, que en nuestro caso serán constantes del tipo “DropdownMenuItem”, y el evento “onChanged”, que imprimirá el valor que escojamos en el drop down.

2.2.5 Checkbox Group

Un Checkbox Group es un conjunto de casillas de verificación (checkboxes) que permite al usuario seleccionar múltiples opciones de una lista. Aquí puedes marcar varias al mismo tiempo.

```
FormBuilderCheckboxGroup(
  decoration:
      const InputDecoration(border: InputBorder.none),
  name: "selectSpeed",
  orientation: OptionsOrientation.vertical,
  // separator: const Padding(padding: EdgeInsets.all(20)),
  options: const [
    FormBuilderFieldOption(value: '20km/h'),
    FormBuilderFieldOption(value: '30km/h'),
    FormBuilderFieldOption(value: '40km/h'),
    FormBuilderFieldOption(value: '50km/h'),
  ],
  onChanged: (List<String>? value) {
    debugPrint(value.toString());
  },
), // FormBuilderCheckboxGroup
```

Para construirlo, deberemos de instanciar la clase “FormBuilderCheckboxGroup” y dentro de esta definir el nombre, la orientación, las opciones (las cuales serán constantes de tipo “FormBuilderFieldOption”) y el manejo del evento “onChanged”.

2.2.6 Floating Action Button

Este botón tendrá la función de mostrar un “AlertDialog” recopilando los datos de todos controles ya creados.

```
floatingActionButton: FloatingActionButton(
    child: Icon(Icons.upload),
    onPressed: () {
      _formKey.currentState?.saveAndValidate();
      String? formString = _formKey.currentState?.value.toStrin
      alertDialog(context, formString!);
    }), // FloatingActionButton
```

Aquí lo que hacemos es guardar y validar el estado actual del formulario, para luego convertir en un mapa de valores el estado actual del formulario.

2.2.7 Demo

Con esto, ya habremos acabado nuestro formulario. Si deseas ver una demo de este lo encontraras en el apartado de **Recursos > Demo form_a**

2.3 Formulario 3: Construcción de un formulario con paquetes adicionales.

2.3.1 Estructura inicial

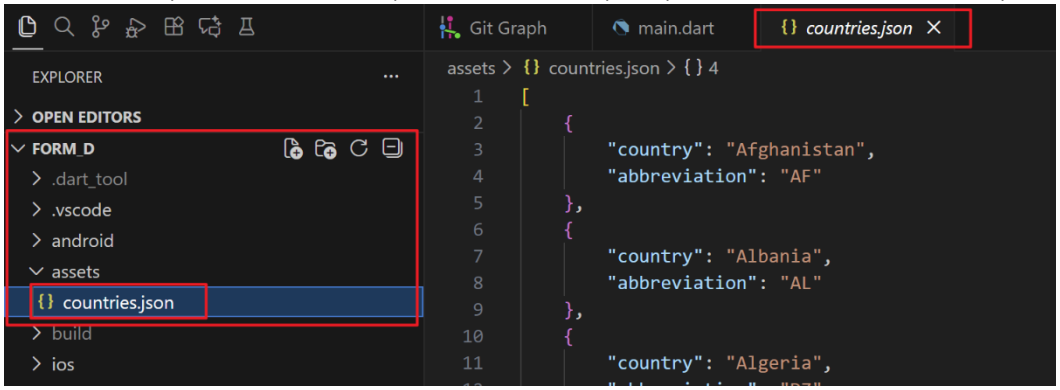
Antes que nada, configuraremos el archivo “pubspec.yaml” para agregar la referencia a nuestra librería.

```
dependencies:
  flutter:
    sdk: flutter

  # The following adds the Cupertino Icons font to your application.
  # Use with the CupertinoIcons class for iOS style icons.
  flutter_form_builder: ^10.2.0
  # Added
  form_builder_extra_fields: ^12.1.0
```

Luego ejecutaremos en la terminal el comando:
|| flutter pub get

el cual descargará los packages que estén declarados dentro del archivo pubspec.yaml. Además, crearemos una carpeta en la raíz llamada “assets” la cual contendrá un archivo .json con todos los países del mundo (este nos será útil para para crear nuestro autocompletador).



Una vez hecho esto, solo faltará importar las librerías dentro de nuestro archivo main.dart.

```dart
import 'package:flutter/material.dart';
import 'package:flutter_form_builder/flutter_form_builder.dart';
import 'package:form_builder_extra_fields/form_builder_extra_fields.dart';
import 'dart:convert';
import 'package:flutter/services.dart' show rootBundle;
```

Además, importaremos la clase rootBundle, la cual nos ayudará a acceder al contenido de nuestra carpeta “assets”.

Con este punto de partida, replicaremos la estructura de nuestro anterior programa haciendo algunas modificaciones extras.

2.3.2 Type Ahead

Este widget es un campo de texto avanzado que proporciona sugerencias a medida que el usuario escribe, permitiendo seleccionar un valor de una lista predefinida o cargada dinámicamente. Es ideal para listas grandes (como países, ciudades o productos).

```dart
class _MyHomePageState extends State<MyHomePage> {
  final String title = 'Salesians Sarrià 25/26 - Damian Altamirano';
  final _formKey = GlobalKey<FormBuilderState>();
  final List<String> _chipOptions = [
    'HTML',
    'CSS',
    'React',
    'Dart',
    'TypeScript',
    'Angular',
  ];
  // 1. Estado para manejar la carga y la lista de nombres
  List<String> countries = [];
  bool isLoading = true;

  @override
  void initState() {
    super.initState();
    _loadData(); // Inicia la carga de datos
  }
```

Antes de usar el componente, es crucial cargar la lista de opciones. Esto se hace en el estado (_MyHomePageState) utilizando el método de ciclo de vida **initState** y una función asíncrona **_loadData()** para leer el archivo JSON (assets/countries.json).

```dart
Future<void> _loadData() async {
  // Lee el archivo JSON
  final String jsonString = await rootBundle.loadString(
    'assets/countries.json',
  );
  final List<dynamic> jsonList = json.decode(jsonString);

  // Mapea para extraer SÓLO la propiedad "country" (el nombre)
  final List<String> loadedNames = jsonList
      .map((json) => json['country'] as String)
      .toList();

  setState(() {
    countries = loadedNames;
    isLoading = false;
  });
}
```

Este método extra todos los valores de la key “country” del JSON y retorna una lista de estos.

```dart
// 3. Función de sugerencias para el TypeAhead
Future<List<String>> _getCountrySuggestions(String pattern) async {
  if (pattern.isEmpty) return countries.take(10).toList();

  // Filtra la lista cargada (countries)
  return countries
      .where((name) => name.toLowerCase().startsWith(pattern.toLowerCase()))
      .toList();
}
```

Esta es la función clave de sugerencias. Filtra la lista countries para devolver solo los nombres que comienzan con el texto ingresado (pattern), mejorando la experiencia de usuario.

Para construir el componente de autocompletado se llama a FormBuilderTypeAhead<String>. Se le asigna un name y se le vinculan dos propiedades esenciales para el autocompletado:

```dart
FormBuilderTypeAhead<String>(
  name: 'country_name',
  decoration: const InputDecoration(
    labelText: 'autocomplete',
    border: OutlineInputBorder(),
  ), // InputDecoration

  // 3. Usamos la función de sugerencias de Strings
  suggestionsCallback: _getCountrySuggestions,

  // 4. itemBuiler para Strings (mucho más simple)
  itemBuilder: (context, String suggestion) {
    return ListTile(title: Text(suggestion));
  },
), // FormBuilderTypeAhead
```

“suggestionsCallback” es la propiedad que se vincula a la función “_getCountrySuggestions”. Esta propiedad se llama cada vez que el texto cambia para solicitar la lista filtrada de sugerencias.

“_itemBuilder” es una función *builder* que define el diseño visual de cada elemento de la lista desplegable de sugerencias, tomando el valor (suggestion como String) y devolviendo un widget (en este caso, un ListTile).

2.3.3 Date Time Picker - Date

Este es un widget que permite a los usuarios seleccionar una fecha, utilizando el selector nativo del sistema operativo. Es usado para manejar la entrada de valores de tiempo precisos dentro de un formulario.

```
FormBuilderDateTimePicker(
  name: 'Date',
  inputType: InputType.date,
  decoration: const InputDecoration(
    labelText: 'Date Picker',
    border: OutlineInputBorder(),
  ), // InputDecoration
), // FormBuilderDateTimePicker
```

Para construir este widget se llama a FormBuilderDateTimePicker, donde se puede asignar el nombre, definir el tipo de entrada como InputType.date, y personalizar su aspecto visual usando la propiedad “decoration”.

2.3.4 Date Range Picker

Se utiliza FormLabelGroup para etiquetar el grupo. El objeto FormBuilderDateRangePicker permite seleccionar un rango de fechas (DateTimeRange). Podemos definir los límites de selección usando firstDate y lastDate, y establecer un valor inicial con initialValue.

```
FormBuilderDateRangePicker(
  name: 'Date Picker',
  decoration: const InputDecoration(
    labelText: 'Date Range',
    hintText: 'Select a range',
    border: OutlineInputBorder(),
  ), // InputDecoration
  firstDate: DateTime(2000),
  lastDate: DateTime(2100),
  initialValue: DateTimeRange(
    start: DateTime.now(),
    end: DateTime.now().add(const Duration(days: 7)),
  ), // DateTimeRange
), // FormBuilderDateRangePicker
```

2.3.5 Date Time Picker – Time

Este es un widget que permite a los usuarios seleccionar una hora, utilizando el selector nativo del sistema operativo. Es usado para manejar la entrada de valores de tiempo precisos dentro de un formulario.

```
FormBuilderDateTimePicker(
  name: 'Time',
  inputType: InputType.time,
  decoration: const InputDecoration(
    labelText: 'Choose a Time',
    border: OutlineInputBorder(),
  ), // InputDecoration
), // FormBuilderDateTimePicker
```

Para construir este widget se llama a FormBuilderDateTimePicker, donde se puede asignar el nombre, definir el tipo de entrada como InputType.time, y personalizar su aspecto visual usando la propiedad “decoration”.

2.3.6 Filter Chips

Este widget permite a los usuarios seleccionar múltiples opciones de un conjunto, ya que no son mutuamente excluyentes (a diferencia de los botones de radio). El resultado es un List de los valores seleccionados.

```
FormBuilderFilterChips(
  name: 'Skills',
  options: _chipOptions
      .map(
        (chip) => FormBuilderChipOption(
          value: chip,
          child: Text(chip),
        ), // FormBuilderChipOption
      )
      .toList(),
  spacing: 8.0,
  runSpacing: 8.0,
), // FormBuilderFilterChips
```

Para construir este objeto, llamamos a FormBuilderFilterChips, definimos el name y le asignamos una lista de FormBuilderChipOption a través de la propiedad options.

```
final List<String> _chipOptions = [
  'HTML',
  'CSS',
  'React',
  'Dart',
  'TypeScript',
  'Angular',
];
```

2.3.7 Floating Action Button

```
floatingActionButton: FloatingActionButton(
  child: Icon(Icons.upload),
  onPressed: () {
    _formKey.currentState?.saveAndValidate();
    String? formString = _formKey.currentState?.value.toString();
    alertDialog(context, formString!);
  },
), // FloatingActionButton
```

Este botón tendrá la función de mostrar un “AlertDialog” recopilando los datos de todos controles ya creados.

Aquí lo que hacemos es guardar y validar el estado actual del formulario, para luego convertir en un mapa de valores el estado actual del formulario.

```
void alertDialog(BuildContext context, String contentText) {
  showDialog<String>(
    context: context,
    builder: (BuildContext context) => AlertDialog(
      title: const Text("Submission Completed"),
      icon: const Icon(Icons.check_circle),
      content: Text(contentText),
      actions: <Widget>[
        TextButton(
          onPressed: () => Navigator.pop(context, 'Tancar'),
          child: const Text('Tancar'),
        ), // TextButton
      ], // <Widget>[]
    ), // AlertDialog
  );
}
```

2.3.8 Demo

Con esto, ya habremos acabado nuestro formulario. Si deseas ver una demo de este lo encontraras en el apartado de **Recursos > Demo form_d**

Recursos

Demo Actividad_01

Enlace:

- actividad_01

Demo form_a

Enlace:

- form_a

Demo form_d

Enlace:

- form_d

Repositorio GitHub

Enlace:

- Mobile_projects_2526