Ficha de Laboratório Nº 7: Formulários, CRUD e REST API

Aplicações Móveis e Serviços - Escola Superior de Tecnologia de Setúbal
2021/2022

Prof. Cédric Grueau
Eng. Filipe Mariano

1. Laboratório Avaliado

1.1. Introdução

O objetivo deste laboratório passa por consolidar e avaliar os conhecimentos adquiridos até agora, nomeadamente nos seguintes aspetos:

- Utilização de Futures para uma programação assíncrona;
- Listagens recorrendo a ListView;
- Navegação entre ecrãs;
- Formulários;

1.2. Objetivo

Produzir uma aplicação simples mas que permita realizar as operações de CRUD numa entidade.

Ao longo das aulas de laboratório, adquiriram o conhecimento suficiente para realizar as diversas operações de CRUD numa entidade, sendo que neste laboratório será aprofundado um pouco mais acerca dos formulários.

2. Formulários

2.1. Tipos de Input/Widgets

2.1.1. Texto

Os *input* do tipo texto são os mais comuns de encontrar em formulários. Este tuoi de Widgets aceitam qualquer texto livre ou que pode ser delimitado pelo tamanho máximo pretendido. São também comuns de utilizar para escrever informação privada, como por exemplo *passwords* ou informação acerca de cartões de crédito.

É possível também costumizar os *input* para se parecerem como *multiline*.

Outros tipos de dados também possíveis de especificar nestes campos de texto, são os numéricos ou outros tipos "especiais" como por exemplo email.

Exemplo de Caixa de Texto:

```
TextField(
  decoration: InputDecoration(
    icon: Icon(Icons.text_format),
    labelText: "Name",
  ),
)
```

2.1.2. Escolha única ou múltipla

Este tipo de *input* permite que os utilizadores selecionem entre uma ou mais opções de escolha, normalmente através de controlos como *checkbox* ou *radio button list*.

A *checkbox* necessita de ter obrigatoriamente a propriedade `value` preenchida. Existe a possibilidade de utilizar o Widget `Checkbox`, que representa uma Checkbox sem texto, ou se pretender adicionar texto, deverá utilizar o Widget `CheckboxListTile`.

Exemplo de *Checkbox*:

```
Checkbox(
  value: false,
  onChanged: (value) {},
)
```

Exemplo de *CheckboxListTile*:

```
CheckboxListTile(
  title: Text('Checkbox com valor a falso de inicio'),
  value: false,
  onChanged: (value) {},
)
```

O mesmo que foi explicado para a *checkbox* também ocorre no caso dos *radio button*, que podem ser representados apenas pelo Widget `Radio` ou em caso de se pretender adicionar texto, deverá utilizar o Widget `RadioListTile`. No caso dos *radio button* é importante referir que a propriedade `value` ao contrário da *checkbox* guarda um valor de qualquer tipo enquanto a propriedade `groupValue` permite indicar qual a *checkbox* que está selecionada, desde que o *groupValue* e o *value* coincidam em termos de valor.

Exemplo de *Radio*:

```
Radio(
  value: false,
  groupValue: false,
  onChanged: (value) {},
)
```

Exemplo de *RadioListTile*:

```
RadioListTile(
  title: Text('Apples'),
  value: 'apples',
  groupValue: 'apples',
  onChanged: (value) {},
),
RadioListTile(
  title: Text('Oranges'),
  value: 'oranges',
  onChanged: (value) {},
)
```

2.1.3. Escolha entre um Conjunto de Valores Pré-definido

Os Widgets que o Flutter disponibiliza para realizar escolhas entre conjuntos de valores pré-definidos, são o `Slider` e a `Dropdown`.

O `Slider` é um tipo de Widget que permite a seleção de um intervalo, definindo um valor mínimo e máximo. O que significa que o intervalo é definido por valores numéricos. Existem ainda as propriedades `divisions` e `label`, em que a primeira permite tornar visível as várias hipóteses de escolha através de uma divisória que aprecerá para cada possibilidade de escolha, enquanto o segundo permite mostrar o valor quando se está a mudar a opção selecionada.

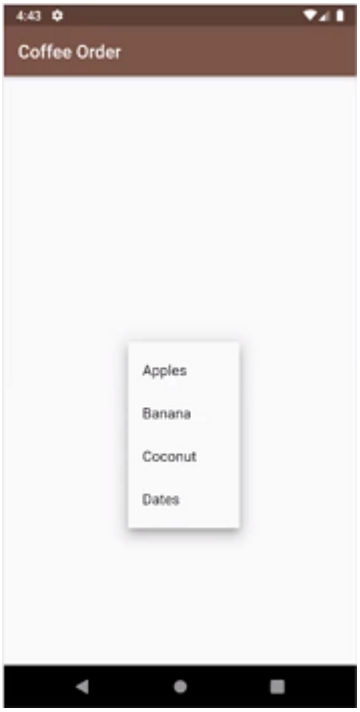
Exemplo de *Slider*:

```
Slider(
  min: 1,
  max: 6,
  value: 2,
  onChanged: (value) {},
)
```

Por sua vez, a `Dropdown` permite uma seleção de um valor a partir de um conjunto de valores pré-definidos, arbitrário e que são do tipo texto. A propriedade `value` permite identificar qual o valor de cada *item*, sendo que a propriedade `items` permite definir as várias opções de escolha na *dropdown*.

Exemplo de *DropdownButton*:

```
var dropdownItems = <String>['Apples','Banana','Coconut', 'Dates'];
var dropdownValue = 'Apples';

DropdownButton(
  value: 2,
  onChanged: (value) {
    setState(() {
      dropdownValue = value;
    });
  },
  items: dropdownItems.map((item) {
    return DropdownMenuItem(
      value: item,
      child: Text(item),
    );
  }).toList(),
)
```

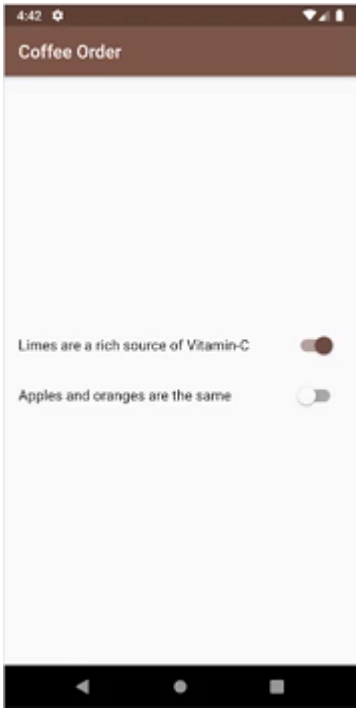


2.1.4. Escolha de Sim/Não ou Verdadeiro/Falso

Este tipo de *input* normalmente serve para representar uma situação de apenas dois valores, como por exemplo uma situação de Sim ou Não, ou de Verdadeito ou Falso.

Exemplo de *SwitchListTile*:

```
SwitchListTile(
  title: Text('Limes are a rich source of Vitamin-C'),
  value: true,
  onChanged: (value) {},
),
SwitchListTile(
  title: Text('Apples and oranges are the same'),
  value: false,
  onChanged: (value) {},
),
```



2.2. Agrupar Widgets de Input

O agrupamento de Widgets de *input* tem como principal objetivo:

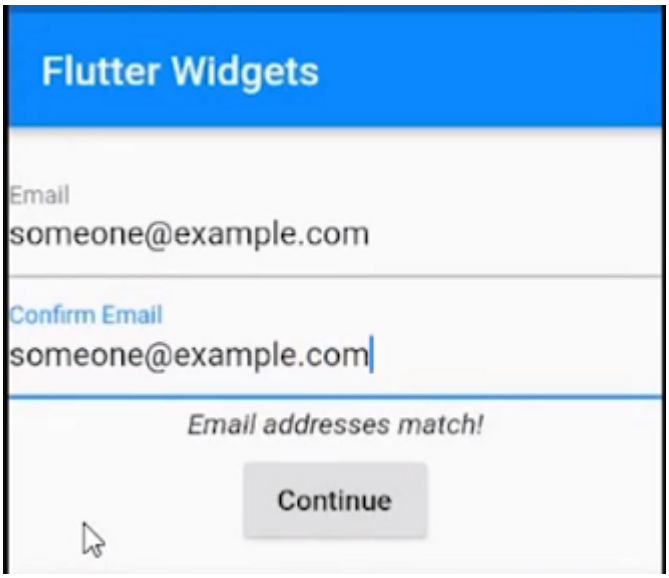
- Reutilização: Pode ser utilizado várias vezes ao longo da aplicação.
- Ligação entre Widgets: Permite interligação entre Widgets mediante o estado ou a passagem de mensagens entre Widgets.
- Dependência: Esses Widgets funcionam como uma única unidade.

Os *inputs* têm constrangimentos e validações associadas a relações que têm em conjunto na representação de um formulário. Por exemplo, um formulário que tenha uma caixa de texto para colocação do email e que peça a confirmação do mesmo, representa duas caixas de texto para colocação do email que deveriam ser agrupadas, geridas e validadas em simultâneo.

Exemplo:

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

class EmailsInput extends StatefulWidget {
  @override
  State<StatefulWidget> createState() => _EmailsInput();
}

class _EmailsInput extends State<EmailsInput> {
  bool match = false;
  String email = "";
  String confirmEmail = "";

  void checkEmails() {
    if (this.email.length > 0 &&
        this.confirmEmail.length > 0 &&
        this.email == this.confirmEmail) {
```

```
      setState(() {
        match = true;
      });
    } else {
      setState(() {
        match = false;
      });
    }
  }

  @override
  Widget build(BuildContext context) {
    return Container(
      child: Column(
        children: [
          TextField(
            decoration: InputDecoration(labelText: "Email"),
            keyboardType: TextInputType.emailAddress,
            onChanged: (String email) {
              setState(() {
                this.email = email;
              });
              checkEmails();
            },
          ),
          TextField(
            decoration: InputDecoration(labelText: "Confirm Email"),
            keyboardType: TextInputType.emailAddress,
            onChanged: (String email) {
              setState(() {
                this.confirmEmail = email;
              });
              checkEmails();
            },
          ),
          Padding(
            padding: EdgeInsets.all(5),
            child: Text(
              this.match
                  ? "Email addresses match!"
                  : "Email addresses do not match",
              style: TextStyle(fontStyle: FontStyle.italic),
            ),
          ),
        ],
      ),
    );
  }
}
```



2.3. Gerir as Alterações nos Inputs

A gestão dos valores nos *inputs* pode ser feita baseado em eventos do próprio Widget, como o evento `onChanged`, ou pode ser feito através de um `Controller` que permite ter maior controlo do *input* pois permite controlar a alteração do valor como também definir um valor inicial.

Por exemplo, nalgumas situações pode ser útil executar uma *callback* sempre que o texto mude numa caixa de texto. Noutras linguagens, normalmente isso é feito através de um evento associado ao componente de *input*. No Flutter existem duas formas de o fazer:

- Fornecendo uma função de *callback* no evento de `onChanged`.
- Utilizando um `TextEditingController`.

A abordagem mais simples é através do evento `onChanged`, em que se pode invocar uma função de *callback* quando ocorrem alterações nesse Widget.

Exemplo:

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

Uma abordagem mais elaborada, mas ao mesmo tempo mais robusta, é a de criar um `controller` associado ao Widget, que permitirá notificar a aplicação quando ocrrem alterações nesse Widget. Funciona através da criação de um `listener` associado ao Widget que se pretende controlar.

Exemplo:

```
class MyCustomForm extends StatefulWidget {
  @override
  _MyCustomFormState createState() => _MyCustomFormState();
```

```
}

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

  void _printLatestValue() {
    print('Text field (controller): ${myController.text}');
  }

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

    myController.addListener(_printLatestValue);
  }

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

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Retrieve Text Input'),
      ),
      body: Padding(
        padding: const EdgeInsets.all(16.0),
        child: Column(
          children: <Widget>[
            TextField(
              onChanged: (text) {
                print('Text field (onChanged): $text'); //utilizando o onChanged
              },
            ),
            TextField(
              controller: myController, //utilizando controller
            ),
          ],
        ),
      ),
    );
  }
}
```

2.4. Widget: Form

O Widget `Form` é um StatefulWidget que funciona como um *container* para agrupar e validar múltiplos campos de um formulário. A forma recomendada de aceder a um formulário é recorrendo a uma chave única que identifica cada formulário e que é gerada pelo Flutter, chamada `GlobalKey`.

Exemplo de GlobalKey:

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

class MyCustomForm extends StatefulWidget {
  @override
  MyCustomFormState createState() {
    return MyCustomFormState();
  }
}

class MyCustomFormState extends State<MyCustomForm> {
  final formKey = GlobalKey<FormState>();

  @override
  Widget build(BuildContext context) {
    return Form(
      key: formKey,
      child: Column(
        children: <Widget>[

        ],
      ),
    );
  }
}
```

Nota: Também é possível utilizar o `Form.of()` para aceder ao formulário a partir de Widgets na árvore desse formulário.

A necessidade de identificar o formulário, prende-se pelo facto de poder realizar operações no mesmo. Nomeadamente de executar a função `save()` que permite invocar o evento de `onSaved` existente no Widget Form.

De seguida será demonstrado um exemplo de como aceder ao formulário a partir da GlobalKey, assim como proceder à gravação do formulário ao clicar num botão. Essa gravação apenas fará a impressão na consola do valor colocado numa caixa de texto.

Exemplo:

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

class BasicForm extends StatefulWidget {
  @override
  State<StatefulWidget> createState() => _BasicFormState();
}

class _BasicFormState extends State<BasicForm> {
  var formKey = GlobalKey<FormState>();
```

```
  @override
  Widget build(BuildContext context) {
    return Container(
      child: Column(
        children: [
          Form(
              key: formKey,
              child: Column(
                children: [
                  TextFormField(
                    onSaved: (String? value) {
                      print("Value: '$value'");
                    },
                  ),
                ],
              )),
          ElevatedButton(
            child: Text("Continue"),
            onPressed: () {
              formKey.currentState!.save();
            },
          )
        ],
      ),
    );
  }
}
```

2.5. Widget FormField e Validação

Os FormField são um tipo de Widgets em que é possível aplicar alguma validação aos dados recebidos.

Para tal, deve ser utilizado a função `validate` desse FormField para conseguir realizar a validação dos dados. Também é possível determinar o modo de validação, ou seja, como e quando ocorre a validação, e que pode ocorrer de duas formas:

- Validação automática: Sempre que há uma alteração no valor do campo, o mesmo é validado (propriedade `autovalidateMode`).
- Validação por ação: Valida o campo apenas quando o utilizador realiza uma ação em que submete o formulário (utilização da função `validator` apenas na submissão do formulário).

No caso dos `TextField` descritos na secção 2.1.1, para validação e em formulários é usual substitui-los por `TextFormField` porque permite definir valores iniciais, assim como aplicar mecanismos de validação nos dados através da função `validator`. Funciona como um *wrapper* do `TextField` num `FormField`.
O `TextField` também permite despoletar alguma função de *callback* no evento de `onChange` mas pode não ir ao encontro do pretendido quando se pretende validar os dados de uma forma mais sistémica, quando por exemplo se submete o formulário.

Resumindo, se for necessário utilizar algum dos campos referidos anteriormente em que seja necessário operações de gravar, limpar ou validar todos os dados inseridos, então deve se recorrer ao `FormField`. Se

apenas for necessário capturar algum valor, o `Field` é suficiente.

O Widget `Form` não é obrigatório que seja utilizado, mas facilita no gravar ou validar todos os campos de uma só vez.

Exemplo da secção anterior (2.4) mas com validação:

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

class BasicForm extends StatefulWidget {
  @override
  State<StatefulWidget> createState() => _BasicFormState();
}

class _BasicFormState extends State<BasicForm> {
  var formKey = GlobalKey<FormState>();

  @override
  Widget build(BuildContext context) {
    return Container(
      child: Column(
        children: [
          Form(
              key: formKey,
              child: Column(
                children: [
                  TextFormField(
                    onSaved: (String? value) {
                      print("Value: '$value'");
                    },
                  ),
                  TextFormField(
                    onSaved: (String? value) {
                      print("Value: '$value'");
                    },
                  ),
                  TextFormField(
                    initialValue: "Hi",
                    onSaved: (String? value) {},
                    validator: (String? value) {
                      if (value?.isEmpty ?? false) {
                        return "Provide a value.";
                      }
                      return null;
                    },
                  ),
                ],
              )),
          ElevatedButton(
            child: Text("Continue"),
            onPressed: () {
              if (formKey.currentState!.validate()) { //validação na ação de
submissão do formulário
```

```
                formKey.currentState!.save();
              }
            },
          )
        ],
      ),
    );
  }
}
```

2.6. Extensão dos FormField

É possível personalizar os próprios FormField de acordo com a aplicação que se está a criar. Para tal, deverá criar os próprios Widgets que serão do tipo FormField e herdando dessa classe as suas características.

No intuito de ver um exemplo deste tipo de personalização, foi disponibilizado um pequeno exemplo com um formulários para caixas de texto do tipo Password, com validação e indicação de modo de validação (ver exemplo disponibilizado no Teams e Moodle).

2.7. Exemplos de Utilização de Formulários

Para visualizar alguns exemplos de utilização de formulários, faça download do rar `flutter_forms` disponibilizado no Teams e Moodle.

3. Exercícios

A entidade que será manipulada através de uma API REST corresponde à descrição de tarefas, associadas a utilizadores e o seu respetivo estado (completadas ou não completadas).

Estrutura da entidade `post`:

Propriedade	Tipo	Constrangimentos
id	number	-
userId	number	Valor entre 1 e 10
title	string	Máximo de 100 caracteres e validar se o campo não está com texto vazio.
body	string	-

API REST a ser utilizada:

https://jsonplaceholder.typicode.com/posts

Guia para a sua utilização:

https://jsonplaceholder.typicode.com/guide/

Nota sobre a API: Para os exercícios que serão realizados a API proposta irá retornar o código 200 (sucesso), mesmo não realizando qualquer operação visto que os dados são estáticos. No entanto, serve apenas para que indiquem se foi feita ou não a operação através da resposta de sucesso do servidor.

Os principais requisitos para a implementação deste laboratório são:

1. Fica ao critério dos alunos os aspetos de *layout* não mencionados e que considerem mais adequados de aplicar.

2. A Home da aplicação deverá mostrar logo a ListView com os resultados provenientes da API (operação de "GET" de tudo).

3. Na Home deverá existir um botão para inserir um novo `post`, que poderá estar como `ElevatedButton` ou uma ação na `AppBar`.

4. Deverá existir ações de `View`, `Edit` e `Delete` associado a um item da ListView.

5. Os ecrãs de `View` e `Edit` poderão ser feitos através de Widgets que simulam um modal (`Dialog`) ou o aparecimento de uma nova *view* controlada pelo `Navigator`.

6. As funções de manipulação da entidade, principalmente de **C**reate, **U**pdate e **D**elete, deverão estar preparadas para que caso a resposta seja de sucesso mostrem informação da operação realizada, como por exemplo uma mensagem de sucesso indicando o `id` em que ocorreu a operação. Em caso de insucesso deverá ser lançada uma excepção. Considere a utilização do package toast ou do snackbar do Scaffold.

7. Os formulários utilizados deverão aplicar as validações referentes aos constrangimentos mencionados.