|  |  |  |
| --- | --- | --- |
|  |  | CASO PRÁCTICO  DISEÑO DE INTERFACES WEB |

# 

# ÍNDICE

[EXPOSICIÓN CASO PRÁCTICO 3](#_Toc110677090)

[DELIVERY APP 5](#_Toc110677091)

[INSTALACIONES NECESARIAS 5](#_Toc110677092)

[CONFIGURACIÓN DEL PROYECTO 7](#_Toc110677093)

[PÁGINA WELCOME 9](#_Toc110677094)

[UNIDADES RELATIVAS 11](#_Toc110677095)

[MODELO DE DATOS: USER 12](#_Toc110677096)

[CONFIGURACIONES PARA LA API 13](#_Toc110677097)

[CONFIGURACIONES PARA LAS PREFERENCIAS 13](#_Toc110677098)

[PÁGINA LOGIN 14](#_Toc110677099)

[PÁGINA BACKBUTTON 21](#_Toc110677100)

[PÁGINA LOADER 22](#_Toc110677101)

[PÁGINA SNACKBAR 22](#_Toc110677102)

[EXPLICACIÓN LÓGICA LOGIN 23](#_Toc110677103)

[USER PROVIDER 23](#_Toc110677104)

[PÁGINA LOGIN PROVIDER 24](#_Toc110677105)

[MODIFICAR MAIN 27](#_Toc110677106)

[COMPROBAR EL LOGIN 28](#_Toc110677107)

[ESQUEMA PASOS LOGIN 29](#_Toc110677108)

[PÁGINA REGISTRO 29](#_Toc110677109)

[PÁGINA INPUT 33](#_Toc110677110)

[EXPLICACIÓN LÓGICA REGISTRO 34](#_Toc110677111)

[MODELO DE DATOS: CLIENTE 36](#_Toc110677112)

[MODIFICACIÓN RETORNO LOGIN RECIBE ID CLIENTE 37](#_Toc110677113)

[MODELO DATOS: DIRECCIÓN 38](#_Toc110677114)

[PÁGINA COMENZAR PEDIDO 41](#_Toc110677115)

[EXPLICACIÓN LÓGICA COMENZAR PEDIDO 45](#_Toc110677116)

[MODIFICAR USER PREFERENCES 45](#_Toc110677117)

[PROVIDER DIRECCIÓN 46](#_Toc110677118)

[MODIFICAR PÁGINA COMENZAR PEDIDO 48](#_Toc110677119)

[PÁGINA ADDDIRECCION 49](#_Toc110677120)

[MODIFICAR PÁGINA DIRECCIONES PROVIDER 54](#_Toc110677121)

[PARA PRÁCTICAR 55](#_Toc110677122)

# EXPOSICIÓN CASO PRÁCTICO

Hasta este punto del curso hemos ido haciendo un ejemplo para ver el funcionamiento básico de Flutter, para crear una aplicación de Flutter que permitiera crear un formulario, validarlo, pasar información entre páginas, etc.

A partir de este momento, iremos adquiriendo nuevos conocimientos y afianzando los que hemos obtenido realizando un caso práctico.

Vamos a suponer que nos han pedido que desarrollemos una Aplicación para poder realizar pedidos en diferentes tipos de restaurantes, de forma que el cliente pueda hacer un único pedido en el que pueda elegir platos de diferentes restaurantes. A partir de este momento llamaremos a nuestra aplicación **DELIVERY APP**.

Para completar las funcionalidades usaremos la **API Delivery** creada en la asignatura *Desarrollo Web en Entorno Servidor*, en el apartado de *Symfony*.

La aplicación arrancará con una *Pantalla Principal* y al pulsar el botón ENTRAR accederemos a una ventana de *Bienvenida* para poder iniciar sesión. Si ya tenemos un email y una contraseña podemos identificarnos y pasar a la siguiente ventana. Si no tenemos cuenta podemos pulsar sobre la palabra *Registro* y pasará a otra página para la *Creación de Cuenta Nueva*. Una vez creada volverá a la pantalla de Bienvenida para iniciar la sesión.

|  |  |  |
| --- | --- | --- |
| Imagen que contiene interior, comida, alimentos, tabla  Descripción generada automáticamente | Interfaz de usuario gráfica, Aplicación  Descripción generada automáticamente | Interfaz de usuario gráfica, Texto, Aplicación  Descripción generada automáticamente |

Una vez identificados pasará a la ventana de *Comenzar Pedido*. En esta página se puede crear una nueva dirección de envío pulsando el texto *Crear nueva dirección* pasando a la ventana *Nueva Dirección*.

|  |  |  |
| --- | --- | --- |
|  |  |  |

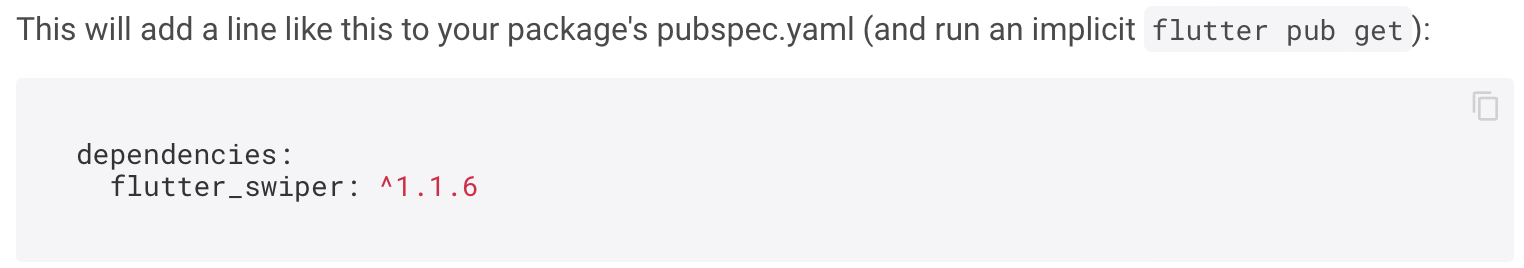
# DELIVERY APP

Creamos un nuevo proyecto llamado **delivery\_app** desde ***Ver🡪Paleta de comandos***.

## INSTALACIONES NECESARIAS

Lo primero que haremos será configurar el proyecto instalando los plugins necesarios.

Desde el archivo *pubspec.yaml* instalaremos el plugin **flutter\_swiper**, que nos permitirá hacer un carrusel de imágenes.



También instalaremos el plugin de **http**, que contiene un conjunto de funciones y clases de alto nivel que facilitan el consumo de recursos HTTP.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Necesitaremos también el plugin **form\_validator,** proporciona un conjunto de validadores integrados y, al mismo tiempo, proporciona un mecanismo para que las aplicaciones proporcionen sus propios validadores. Todos los validadores integrados se pueden deserializar a través de JSON.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente con confianza media

El plugin **provider**, como gestor de estados para no tener que utilizar la forma normal de Flutter para pasar información entre diferentes Widgets.

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza media

El plugin **shared\_preferences**. El uso principal de *Shared Preferences* es guardar las preferencias de los usuarios, configuraciones, tal vez datos (si no es muy grande) para que la próxima vez que la aplicación sea lanzada, estas piezas de información puedan ser recibidas y usadas.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

El plugin **intl**, es una extensión que facilita la gestión entre las traducciones de archivos .arb y la aplicación Flutter. Genera el código necesario para usar la biblioteca oficial Dart Intl y autocompleta las claves de traducción en el código. También sirve para gestionar las fechas.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

El plugin **modal\_bottom\_sheet**, que son menús emergentes que se nos suelen desplegar desde la parte inferior de la pantalla y que son muy usados para incluir en ellos diferentes opciones de compartir o simplemente un listado de subopciones a realizar sobre un elemento en pantalla.

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza media

Para que nos muestre el panel de archivos con iconos podemos instalar la extensión **Material Icon Theme**.

## CONFIGURACIÓN DEL PROYECTO

|  |  |
| --- | --- |
| Vamos a hacer una estructura más organizada que en el proyecto anterior que utilizamos de ejemplo para ir aprendiendo Flutter.  Vamos a abrir el **assets** del archivo*pubspec.yaml* y cogerá todo lo que tengamos dentro de assets:  assets:  - assets/images/  Creamos las carpetas dentro del proyecto | Interfaz de usuario gráfica, Texto, Aplicación  Descripción generada automáticamente |

Dentro de la carpeta **lib** crearemos una carpeta llamada **config** y dentro de ella una carpeta llamada **colors**. En esta carpeta crearemos un archivo llamado *colors.dart,* en este archivo crearemos variables que nos den un color predefinido que vamos a utilizar a menudo dentro de la aplicación.

import 'package:flutter/material.dart';

final amarillo = Color.fromRGBO(255, 204, 0, 1.0);

final gris = Color.fromRGBO(184, 187, 198, 1.0);

final rosa = Color.fromRGBO(255, 45, 85, 1.0);

final bgGris = Color.fromRGBO(247, 248, 250, 1.0);

En el archivo *main.dart* borramos todo lo que aparece a partir de la clase *MyHomePage* y realizamos los siguientes cambios en el código que hemos dejado. Básicamente vamos a configurar los colores que se utilizarán en el tema, quitar la bandera de *debug*, cambiar el título de la aplicación y dejar preparadas las propiedades de ruta y ruta inicial.

import 'package:flutter/material.dart';

void main() {

runApp(const MyApp());

}

class MyApp extends StatelessWidget {

const MyApp({Key? key}) : super(key: key);

// This widget is the root of your application.

@override

Widget build(BuildContext context) {

return MaterialApp(

debugShowCheckedModeBanner: false,

routes: ,

initialRoute: ,

title: 'DELIVERY APP',

theme: ThemeData(

accentColor: Color.fromRGBO(255, 140, 0, 1.0),

primaryColor: Color.fromRGBO(10, 31, 68, 1.0),

buttonColor: Color.fromRGBO(0, 122, 255, 1.0),

disabledColor: Color.fromRGBO(142, 142, 147, 1.2),

scaffoldBackgroundColor: Colors.white,

appBarTheme: AppBarTheme(

iconTheme: IconThemeData(color: Colors.black)

),

),

);

}

}

Ahora vamos a configurar el archivo de rutas. Crearemos una carpeta en **lib🡪config** llamada **routes** y dentro de ella un archivo llamado *routes.dart.*

import 'package:delivery\_app/config/pages/login\_page/login\_page.dart';

import 'package:delivery\_app/config/pages/welcome\_page/welcome\_pages.dart';

import 'package:flutter/widgets.dart';

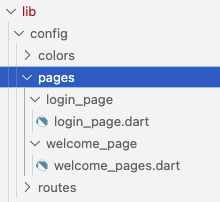
final routes = <String, WidgetBuilder>{

'welcome': (BuildContext context) => WelcomePage(),

'login': (BuildContext context) => LoginPage()

};

Previamente habremos creado la siguiente estructura de carpetas y archivos utilizando el esqueleto **statelessW** en ambas páginas:



Volvemos a *main.dart* y en **routes** llamamos *routes* y el **initialRoute** será ‘*welcome’*.

routes: routes,

initialRoute: 'welcome',

## PÁGINA WELCOME

En la página de **WelcomePage** vamos a poner un texto para comprobar que funciona.

Arrastramos las imágenes del fondo de la página de Welcome y de Login a la carpeta **assets🡪images.**

Vamos a la página de *WelcomePage* y cambiamos el **Container** por un **Scaffold.**

@override

Widget build(BuildContext context) {

return Scaffold(

En *body* ponemos un Widget **Stack**, que es un contenedor que sirve para poner contenedores uno dentro de otro.

body: Stack(

**Stack** tiene un array de Widgets que se van a ir mostrando unos dentro de otros.

children: [

Ponemos un contenedor normal con un **BoxDecoration** para coger la imagen y ponerla de fondo al contenedor. Tiene dos propiedades, *image* para poner la imagen y *child* para poner “algo” encima de la imagen. En nuestro caso pondremos una capa negra con una opacidad, para hacerla semitransparente y que se vea mejor.

Container(

decoration: BoxDecoration(

image: DecorationImage(

Con esto hacemos que se adapte lo mejor posible al teléfono y que lo cubra sin deformarla.

fit: BoxFit.cover,

image: AssetImage('assets/images/fondo-Welcome.jpeg')

)

),

child: BackdropFilter(

El *blur* es como un difuminado.

filter: ImageFilter.blur(sigmaX: 1.0, sigmaY: 1.0),

child: Container(

color: Colors.black.withOpacity(0.3),

),

),

),

Ahora crearemos una columna donde haremos que los widgets aparezcan uno debajo de otro.

Column(

Con *mainAxisAligment.center* hacemos que los elementos estén centrados verticalmente.

mainAxisAlignment: MainAxisAlignment.center,

Con crossAxisAlignment hacemos que los elementos esten centrados horizontalmente. Como es un contenedor de tipo columna su eje principal es el vertical y si eje cruzado el horizontal.

crossAxisAlignment: CrossAxisAlignment.center,

children: [

Ponemos un título y le damos estilo.

Container(

padding: EdgeInsets.symmetric(horizontal: 50.0),

child: Text(

'APP DELIVERY',

style: TextStyle(

color: Colors.white,

fontWeight: FontWeight.bold,

fontSize: 45.0),

),

),

Ponemos un subtítulo y le damos estilo.

Container(

padding: EdgeInsets.symmetric(vertical: 30.0),

child: Text(

'La mejor comida en tu casa',

style: TextStyle(

color: Colors.white,

fontSize: 17.0,

fontWeight: FontWeight.w400),

),

),

Ponemos un botón y le damos estilo.

Container(

width: ancho \* 0.9,

height: alto \* 0.06,

child: ElevatedButton(

onPressed: () => {},

style: ElevatedButton.styleFrom(

primary: Theme.of(context).accentColor,

shape: RoundedRectangleBorder(

borderRadius: BorderRadius.circular(20.0))),

child: Text('ENTRAR'),

),

)

],

)

],

),

);

}

}

### UNIDADES RELATIVAS

Como podemos observar el tamaño del botón lo hemos puesto con coordenadas absolutas, lo cual no es nada recomendable porque dependiendo del dispositivo donde se visualice la App puede aparecer de diferente manera. Lo ideal es utilizar unidades relativas. En el caso de Flutter para poder obtener medidas relativas o porcentajes tenemos que hacer uso de las *MediaQueries*.

En el método **build**, al principio, antes del *return Scaffold* introduciremos las siguientes variables:

var ancho = MediaQuery.of(context).size.width;

var alto = MediaQuery.of(context).size.height;

En ellas lo que haremos será guardar el ancho y el alto de la pantalla en la que se está mostrando nuestra aplicación.

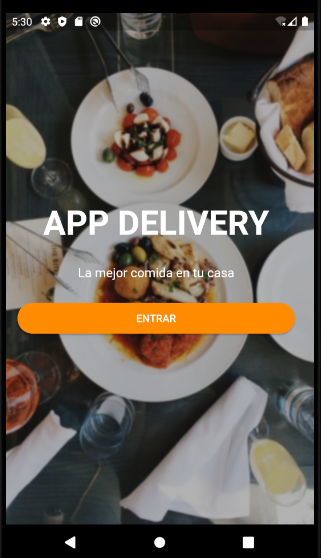
Modificaremos el tamaño del botón de la siguiente manera:

width: ancho \* 0.9,

height: alto \* 0.06,

Ahora vamos a hacer que al pulsar el botón pasemos a la página del *Login*:

onPressed: () => Navigator.pushNamed(context, 'login'),



## MODELO DE DATOS: USER

Vamos a hacer el modelo de datos utilizando el siguiente JSON:

{

"id": 5,

"email": "[prueba@prueba.com](mailto:prueba@prueba.com)",

"roles": [

"ROLE\_CLIENTE",

"ROLE\_USER"  
]

}

Lo copiamos y desde **Ver🡪Paleta de comandos** buscamos *JSON to Dart: From Clipboard*. Como nombre de la clase le pondremos *user*, al resto de opciones le decimos que no menos a *Implement Map/JSON methods with decode/encode* que le diremos que sí. Como carpeta elegimos **lib🡪models**. Con esto se genera el archivo *user.dart* al que le haremos los siguientes cambios:

import 'dart:convert';

class User {

int? id;

String? email;

List<dynamic>? roles;

String? password;

String? token;

User({this.id, this.email, this.roles, this.password, this.token});

factory User.fromMap(Map<String, dynamic> data) => User(

id: data['id'] as int?,

email: data['email'] as String?,

roles: data['roles'] as List<dynamic>?,

password: data['password'] as String?,

token: data['token'] as String?

);

Map<String, dynamic> toMap() => {

'id': id,

'email': email,

'roles': roles,

'password': password,

'token': token

};

/// `dart:convert`

///

/// Parses the string and returns the resulting Json object as [User].

factory User.fromJson(String data) {

return User.fromMap(json.decode(data) as Map<String, dynamic>);

}

/// `dart:convert`

///

/// Converts [User] to a JSON string.

String toJson() => json.encode(toMap());

}

## CONFIGURACIONES PARA LA API

Dentro de la carpeta **config** vamos a crear una nueva carpeta llamada **api** y dentro un archivo llamado *api\_url.dart*. En este archivo crearemos una clase llamada **Api** y pondremos las rutas básicas.

class Api {

static const baseUrl = 'http://127.0.0.1:8000/api';

static const login = baseUrl + '/login\_check';

}

## CONFIGURACIONES PARA LAS PREFERENCIAS

Dentro de la carpeta **config** creamos otra carpeta llamada **preferences** y dentro de esta carpeta crearemos un archivo llamado *shared\_preferences.dart.* Aquí crearemos una clase e iremos poniendo los métodos que necesitemos, por ejemplo, el id del usuario cuando logee, el token del usuario cuando logee, etc. También, podríamos hacer una página de bienvenida para la primera vez que arranque la aplicación que explique cómo funciona, por ejemplo, con una variable de *welcome* a true o false.

import 'package:shared\_preferences/shared\_preferences.dart';

import '../../models/user.dart';

class UserPreferences {

Hacemos un método *saveUserOnLogin* que tiene que ser Future, porque va a ser un método asíncrono y recibirá un *User*.

Future<bool> saveUserOnLogin(User user) async {

Nos traemos una instancia de las SharedPreferences.

final SharedPreferences prefs = await SharedPreferences.getInstance();

Guardamos el *token* como String y el *id* como entero, esta información la lleva el *User*. Le ponemos ! porque sabemos que cuando hacemos un “login” que sea correcto siempre va a devolver un token y un id.

prefs.setString('token', user.token!);

prefs.setInt('User\_id', user.id!);

return true;

}

}

## PÁGINA LOGIN

Vamos a abrir la página de *login\_page.dart*. Cambiamos la extensión de la clase a **Statelessful** (si nos ponemos encima de **StatelessWidget** pulsamos **Ctrl+.** o **Cmd+.** y elegimos la opción *Convert to Stateful).*

class \_LoginPageState extends State<LoginPage> {

Cogemos el **keyForm** del formulario y los controles del email y del password. El **keyFom** será el que le vamos a dar al Widget del formulario para comprobar el estado.

GlobalKey<FormState> keyForm = GlobalKey<FormState>();

TextEditingController emailCtrl = TextEditingController();

TextEditingController passCtrl = TextEditingController();

@override

Widget build(BuildContext context) {

Ponemos las variables ancho y alto para luego utilizar las medidas relativas.

var ancho = MediaQuery.of(context).size.width;

var alto = MediaQuery.of(context).size.height;

Vamos a configurar el *StatusBar* de arriba. Con el método *setSystemUIOverlayStyle* estamos accediendo al tema por defecto del sistema y lo sobreescribiremos con un color transparente y con brillo. Lo que hacemos es que aparezca la barra superior que antes no se veía:

|  |  |
| --- | --- |
| ANTES:  Forma, Rectángulo  Descripción generada automáticamente | DESPUES:  Forma  Descripción generada automáticamente con confianza media |

SystemChrome.setSystemUIOverlayStyle(SystemUiOverlayStyle.light.copyWith(

statusBarColor: Colors.transparent,

statusBarIconBrightness: Brightness.light,

));

return Scaffold(

body: Container(

*SingleChildScrollView* se utiliza para que la pagina pueda hacer scroll cuando se levanta el teclado, por ejemplo, cuando tengamos inputs.

child: SingleChildScrollView(

Para que siempre deje subir y bajar y evitar el efecto “rebote” en la página.

physics: AlwaysScrollableScrollPhysics(),

Utilizamos un *Column* para en un único hijo poder poner varios Widgets.

child: Column(

children: [

*Stack* se utiliza para apilar Widgets encima de otros. Cuando colocamos algo en un *Stack* siempre se coloca arriba a la izquierda.

Stack(

children: [

Image(

Ponemos un ancho que ocupa todo el espacio que da la pantalla.

width: double.infinity,

El alto lo ponemos utilizando medidas relativas.

height: alto \* 0.5,

Con esto hacemos que sea lo más pequeña posible cubriendo todo lo que tiene que cubrir.

fit: BoxFit.cover,

image: AssetImage('assets/images/login-fondo.jpeg'),

Con esto quedaría así:



), //Image

Ahora vamos a disfrazar la parte de abajo para que parezca que está por encima de la foto. Vamos a poner el botón de vuelta manualmente (backButton). Para los widgets que utilizaremos con mucha frecuencia vamos a crear un archivo aparte, dentro de **lib** crearemos una carpeta llamada **Widgets** y dentro un archivo llamada *backButton.dart* que mostramos y explicamos al final del código de *login\_page.dart*. Utilizamos el widget y le pasamos un color blanco y un margen por arriba para separlo.

Container(

child: backButton(Colors.white, context),

margin: EdgeInsets.only(top: 50.0),

|  |  |
| --- | --- |
| Se vería así: |  |

)//Container

],

) //Stack

*Transform.translate* es un Widget que permite a su hijo hacerle un *Offset*, es decir, poner por encima de donde debería estar pintado hacia el lado que queramos.

Transform.translate(

offset: Offset(0.0, -20),

child: Container(

width: double.infinity,

height: alto \* 0.5,

decoration: BoxDecoration(

color: Colors.white,

borderRadius: BorderRadius.circular(20.0)

),

|  |  |
| --- | --- |
| Así es como se vería: | Un par de platos con comida  Descripción generada automáticamente |

Al *Container* le añadimos el hijo, que tendrá un *Padding* general de 20.

child: Padding(

padding: EdgeInsets.all(20.0),

Para poner varios elementos en vertical utilizamos el Widget *Column.* Pondremos varios textos, un formulario y varios enlaces relacionados con contraseña olvidada.

child: Column(

children: [

Text('Bienvenido',

style: TextStyle(

color: Theme.of(context).primaryColor,

fontWeight: FontWeight.bold,

fontSize: 26.0,

)

),

Text('Accede a la cuenta',

style: TextStyle(

color: gris,

fontWeight: FontWeight.w500,

fontSize: 13.0,

)

),

Form(

key: keyForm,

child: Column(

children: [

\_emailInput(),

\_passwordInput(),

\_loginbutton()

],

)

),

Container(

margin: EdgeInsets.only(top: 20.0),

El Widget *GestureDetector* sirve para pulsar sobre un texto porque podemos ponerle un método **onTap**.

child: GestureDetector(

onTap: () => {},

child: Text('Contraseña olvidada?'),

)

),

Container(

margin: EdgeInsets.only(top: 20.0),

child: Row(

mainAxisAlignment: MainAxisAlignment.center,

children: [

Text(

'Aún no tienes cuenta?',

style: TextStyle(

color: gris,

fontWeight: FontWeight.w500,

fontSize: 15.0),

),

Text(

'Registro',

style: TextStyle(

color: Theme.of(context).accentColor,

fontWeight: FontWeight.w400,

fontSize: 15.0),

)

],

),

)

],

),

),

),//Container

) // Transform

],

),

),

),

);

}

Vamos a implementar los métodos que creen los inputs email, password y el botón para Entrar.

Widget \_emailInput() {

return Container(

Le damos la medida de altura, aunque deberíamos hacerlo de forma relativa.

height: 40.0,

margin: EdgeInsets.only(top: 40.0),

padding: EdgeInsets.only(left: 20.0),

Al contenedor le ponemos un borde “redondito”.

decoration: BoxDecoration(

color: Color.fromRGBO(142, 142, 147, 1.2),

borderRadius: BorderRadius.circular(30.0),

),

Aquí ponemos el input, como está dentro de un **Form**, tiene que ser un *TextFormField.*

child: TextFormField(

Para que salga un teclado acorde a los datos de tipo email.

keyboardType: TextInputType.emailAddress,

controller: emailCtrl,

Como instalamos la librería de “validator” comprobamos el dato haciendo uso de ella.

validator:

ValidationBuilder().email('Debe ser un email válido').build(),

decoration: InputDecoration(

hintText: 'Email',

Como ya hemos puesto los bordes en el *decoration* de arriba aquí se los quitamos.

border: OutlineInputBorder(borderSide: BorderSide.none)),

),

);

}

Widget \_passwordInput() {

return Container(

height: 40.0,

margin: EdgeInsets.only(top: 10.0),

padding: EdgeInsets.only(left: 20.0),

decoration: BoxDecoration(

color: Color.fromRGBO(142, 142, 147, 1.2),

borderRadius: BorderRadius.circular(30.0),

),

child: TextFormField(

Para que no salga la contraseña que estamos escribiendo.

obscureText: true,

controller: passCtrl,

validator:

ValidationBuilder().minLength(6, 'Mínimo 6 caracteres').build(),

decoration: InputDecoration(

hintText: 'Contraseña',

border: OutlineInputBorder(borderSide: BorderSide.none)),

),

);

}

Widget \_loginbutton() {

return Container(

width: 350,

height: 40,

margin: EdgeInsets.only(top: 30.0),

child: ElevatedButton(

style: ElevatedButton.styleFrom(

primary: Theme.of(context).accentColor,

shape:

RoundedRectangleBorder(

borderRadius: BorderRadius.circular(20.0)),

),

child: Text('Entrar'),

onPressed: () {

Como no sabemos si le hemos puesto algo a los inputs en *currentState* añadimos el ?. Si el *currentState* no es nulo hará la validación de los campos. Esto devuelve verdadero o falso. Si es falso automáticamente muestra los mensajes de error correspondientes de los inputs. Si es verdadero hacemos un mapa de tipo *String* y *dynamic* y le pasamos el email y la contraseña.

//Validar los inputs

var valid = keyForm.currentState?.validate();

if (valid == true) {

Map<String, dynamic> user = {

'username': emailCtrl.text,

'password': passCtrl.text

};

***En este punto incluiremos la lógica del LOGIN que explicamos más abajo\*.***

Creamos una variable de tipo *LoginProvider* y de momento ponemos el *listen* a **false**, con esto le estamos indicando que no es necesario que el provider escuche. Podemos acceder al usuario pero no modificarlo

LoginProvider loginProvider =

Provider.of<LoginProvider>(context, listen: false);

Hacemos un **final** de tipo *Future<String>* que es lo que devuelve el método login al que le pasamos el user.

final Future<String> text = loginProvider.login(user, context);

Dependiendo del estado, si se está autentificando pondremos un loader, que explicamos más adelante como lo creamos, en el momento que deje de estar autentificando hará un *Navigator.pop* y cerrara el círculo de progreso.

loginProvider.loginSt == LoginStatus.Authenticating

? loading(context)

: Navigator.pop(context);

Cuando acaba la carga cogemos el texto y le hacemos un *then*, cuando ya tenemos el valor de la variable cogemos el valor y comprobamos

text.then((msg) {

Si el estado es no logeado mostramos el mensaje en un *snackbar* que haremos en otro archivo y mostramos abajo

if (loginProvider.loginSt == LoginStatus.NotLoggedIn) {

snackBar(context, msg);

}

});

}

},

),

);

}

}

|  |  |
| --- | --- |
| Así es como se vería: | Interfaz de usuario gráfica, Aplicación  Descripción generada automáticamente |

## PÁGINA BACKBUTTON

Código de la página *backButton.dart*:

import 'package:flutter/material.dart';

Hacemos un método que devuelve un Widget y que recibe un color para personalizarlo y un contexto para que sepa donde se va a mostrar.

Widget backButton(Color color, BuildContext context) {

return IconButton(

Al pulsar hacemos que vuelva a la página posterior con **Navigator.pop**

onPressed: () => Navigator.pop(context),

icon: Icon(Icons.arrow\_back, color: color, size: 28.0));

}

## PÁGINA LOADER

Código de la página *loader.dart*:

import 'package:flutter/material.dart';

loading(BuildContext context) {

Necesitamos el *context* porque se va a mostrar en otro sitio, para saber dónde se va a mostrar.

showDialog(

context: context,

La propiedad *barrierDismissible* a **false** hace que no se cierre si tocamos fuera.

barrierDismissible: false,

builder: (BuildContext context) {

Devuelve un círculo de progreso.

return Center(

child: CircularProgressIndicator(),

);

});

}

## PÁGINA SNACKBAR

Código de la página *snackbar.dart*:

import 'package:flutter/material.dart';

snackBar(BuildContext context, String msg) {

showDialog(

context: context,

builder: (context) {

Lo que hace es que en el momento en que aparezca espera 5 segundos y luego ejecuta lo que hay dentro, en este caso se cerrará.

Future.delayed(Duration(seconds: 5), () {

Navigator.pop(context)

});

Esto será lo que muestre.

return AlertDialog(

title: Text(msg),

);

});

}

## EXPLICACIÓN LÓGICA LOGIN

\* ***Explicación lógica LOGIN***: A continuación, haremos la lógica del ***Login***. Para ello necesitamos utilizar los **providers**, cuya librería instalamos con anterioridad en el *pubspec.yaml*. Un **provider** sirve tanto para llamar las APIs como para notificar a cualquier Widget que este escuchando los cambios sobre una variable que este en el propio **provider,** de forma que cuando cambie esa variable ese Widget se entera y se actualiza con los nuevos datos.

Básicamente un **provider** crea una especie de notificador de cambios y cada vez que hay un cambio notifica a todos los Widgets que están suscritos a ese **provider** para que sepan que ha habido un cambio y que se tienen que redibujar.

Nosotros vamos a utilizar los **providers** tanto para hacer llamadas a las APIS como para controlar los datos que necesitemos.

Vamos a estructurarlo de la siguiente manera, por una parte, tendremos **providers** generales para nuestros modelos de datos (clientes, user, direcciones, pedidos, etc.) y luego haremos **providers** para las llamadas a las APIs.

Primero haremos los **providers** generales y luego los propios de cada página de llamadas a las APIS.

### USER PROVIDER

Lo primero que vamos a hacer será el primer **provider** para el objeto *user*. Creamos dentro de **lib** una carpeta llamada **provider** y dentro un archivo llamado *user\_provider.dart.*

import 'package:flutter/material.dart';

import '../models/user.dart';

Extiende de *ChangeNotifier*, que es quien va a estar notificando a los que estén a la escucha de los cambios producidos.

class UserProvider extends ChangeNotifier {

Tendremos una variable user privada

User \_user = User();

Que tendrá un get

User get user => \_user;

Y un set, que recibirá un usuario.

void setUser(User user) {

\_user = user;

Cuando hagamos *Login* en la respuesta devolverá el **token** y el **id**, en base a esta información haremos un objeto *user* con el modelo que tenemos. Una vez tenemos el objeto le diremos al **provider** que coja al user que tiene y lo “setee” al que tenemos.

En ese momento le diremos *notifyListeners* para que el widget que este a la escucha reciba ese nuevo user. La siguiente pantalla después de hacer *Login* podría ser la lista de las direcciones y allí necesitaría para la API el **id** del usuario. Para sacar ese **id**, al principio podría decirle que tengo una variable de tipo *User* que viene de *UserProvider*. Cuando tenga esa variable podré coger el **id** y llamar. Como sabemos que ya la tiene, en el momento que hagamos *setUser* hará un *notifyListeners* y ese Widget recibirá el usuario.

notifyListeners();

}

}

### PÁGINA LOGIN PROVIDER

El **provider** se puede utilizar para hacer llamadas a las APIS y lo vamos a utilizar porque nos permite crear una variable con el estado de la llamada y en base a ese estado de la llamada en el widget principal mostrar o un loading, o un mensaje de fallo, etc.

Los **providers** propios de las páginas los vamos a crear dentro de la carpeta de la propia página. En nuestro caso haremos dentro de la carpeta **login\_page** haremos una carpeta llamada **provider** y dentro de ella una archivo llamado *login\_provider.dart*

import 'package:flutter/material.dart';

Creamos un enumerado con los estados, porque en base a esos estados el widget que está escuchando hará una acción u otra.

enum LoginStatus { NotLoggedIn, LoggedIn, Authenticating, LoggedOut }

Creamos la clase que extiende de *ChangeNotifier*

class LoginProvider extends ChangeNotifier {

Creamos una variable privada para que guarde el estado del login y luego le haremos un get y un set. Por defecto empieza en el estado no logeado.

LoginStatus \_loginSt = LoginStatus.NotLoggedIn;

LoginStatus get loginSt => \_loginSt;

set loginSt(LoginStatus status) {

\_loginSt = status;

notifyListeners();

}

Vamos a hacer un método para llamar al login de la API. Hay varias formas de hacerlo. Como es un método asíncrono tiene que devolver un Future que en este caso será de tipo String porque dependiendo del Satus Code devolverá un String u otro. Ese String luego lo mostraremos en la aplicación. Este método recibirá el mapa que hemos hecho antes en el login y el contexto porque si el login está ok navegaremos a la página que queramos desde aquí.

Future<String> login(Map<String, dynamic> data, BuildContext context) async {

Primero modificamos la variable *\_loginSt* a estado *autentificación*. Después hacemos un *notifyListeners* y desde el propio Widget pondremos un círculo de progreso para indicar que está haciendo login.

\_loginSt = LoginStatus.Authenticating;

notifyListeners();

Traemos el objeto **Response** que tiene que venir de *http.dart* y tiene que esperar a la respuesta. Haremos un *post*, porque login era un post. Tenemos que pasarle la *url*, los datos que se van a enviar en el *body*, que tienen que pasarse a JSON. Como es un Map automáticamente lo parsea a JSON. A veces falla, entonces en *headers* le decimos que es una aplicación de tipo JSON. Cuando hacemos una petición a API siempre hay que decirle de que tipo es el contenido de la petición, es una medida extra de seguridad.

Podéis encontrar más información de las cabeceras de petición en: <https://developer.mozilla.org/es/docs/Web/HTTP/Headers>

Response response = await post(

Uri.parse(Api.login),

body: json.encode(data),

headers: {

'Content-Type': 'application/json',

}

);

Si la respuesta es 200 significa que esta todo bien.

if (response.statusCode == 200) {

Creamos un usuario con la información recibida por el body.

User user = User.fromJson(response.body);

Guardamos el user en UserPreferences.

UserPreferences().saveUserOnLogin(user);

Cambiamos el estado a logeado y lo notificamos.

\_loginSt = LoginStatus.LoggedIn;

notifyListeners();

Notificamos al UserProvider que tenemos un User, para que a todo el que este esperando un objeto user, se lo pase. El **id** lo sacaremos del *Provider* cada vez que lo necesitemos y el token lo sacaremos del *sharedPreferences.* Avisamos al *UserProvider* para que cambie el **user** por el que he obtenido y como el *setUser* tiene un *notifyListener*, a los widgtes que estén a la espera en ese momento les va a llegar la información del user, obviamente este conlleva que en esos widgets tendremos que controlar si llega el dato o no para que hagan la acción que queramos.

Provider.of<UserProvider>(context, listen: false).setUser(user);

Cuando logeemos iremos a la pantalla de Welcome

Navigator. pushNamed(context, 'welcome');

}

Si la respuesta no ha sido positiva cambiamos el estado a no logeado y lo notificamos. Si el error ocurrido es el 401 decimos que las credenciales no son correctas y para cualquier otro tipo de error damos otro mensaje más general.

\_loginSt = LoginStatus.NotLoggedIn;

notifyListeners();

if (response.statusCode == 401) {

return 'Credenciales incorrectas';

} else {

return 'Ha ocurrido un error';

}

}

}

### MODIFICAR MAIN

Ahora tenemos que decirle al *main.dart* que existen esos providers. Cuando solo tenemos un provider lo pasamos como *Provider*, pero cuando existen muchos tenemos un Widget que se llama *MultiProvider*.

import 'package:delivery\_app/config/pages/login\_page/provider/login\_provider.dart';

import 'package:delivery\_app/config/routes/routes.dart';

import 'package:delivery\_app/providers/user\_provider.dart';

import 'package:flutter/material.dart';

import 'package:provider/provider.dart';

void main() {

runApp(const MyApp());

}

class MyApp extends StatelessWidget {

const MyApp({Key? key}) : super(key: key);

// This widget is the root of your application.

@override

Widget build(BuildContext context) {

return MultiProvider(

El MultiProvider contiene un array de los providers que vamos creando.

providers: [

El \_ dentro de (\_) significa que hace referencia a *BuildContext context,* así nos ahorramos escribirlo.

ChangeNotifierProvider(create: (\_) => LoginProvider()),

ChangeNotifierProvider(create: (\_) => UserProvider())

],

Envolvemos toda la App con un *Multiprovider,* gracias a esto hace que los providers tengan acceso a todos los widgets que están dentro de *MaterialApp,* es decir, se puede acceder a todas las partes de la apliación.

child: MaterialApp(

debugShowCheckedModeBanner: false,

routes: routes,

initialRoute: 'welcome',

title: 'DELIVERY APP',

theme: ThemeData(

accentColor: Color.fromRGBO(255, 140, 0, 1.0),

primaryColor: Color.fromRGBO(10, 31, 68, 1.0),

buttonColor: Color.fromRGBO(0, 122, 255, 1.0),

disabledColor: Color.fromRGBO(142, 142, 147, 1.2),

scaffoldBackgroundColor: Colors.white,

appBarTheme:

AppBarTheme(iconTheme: IconThemeData(color: Colors.black)),

),

),

);

}

}

***\* Una vez completados estos pasos podríamos volver a la página login\_page.dart y completar el proceso de la lógica del Login.***

### COMPROBAR EL LOGIN

Abrimos el Mamp, el PhpStorm y arrancamos el proyecto de **delivery**.

*Para ejecutar en el emulador de Android tenemos que cambiar la dirección base por*

static const baseUrl = 'http://10.0.2.2:8000/api';

*Para ejecutar en Chrome tenemos que hacer un cambio en Symfony, aunque no es seguro. Tenemos que instalar el plugin NelmioCorsBundle con la orden ‘composer req cors’.*

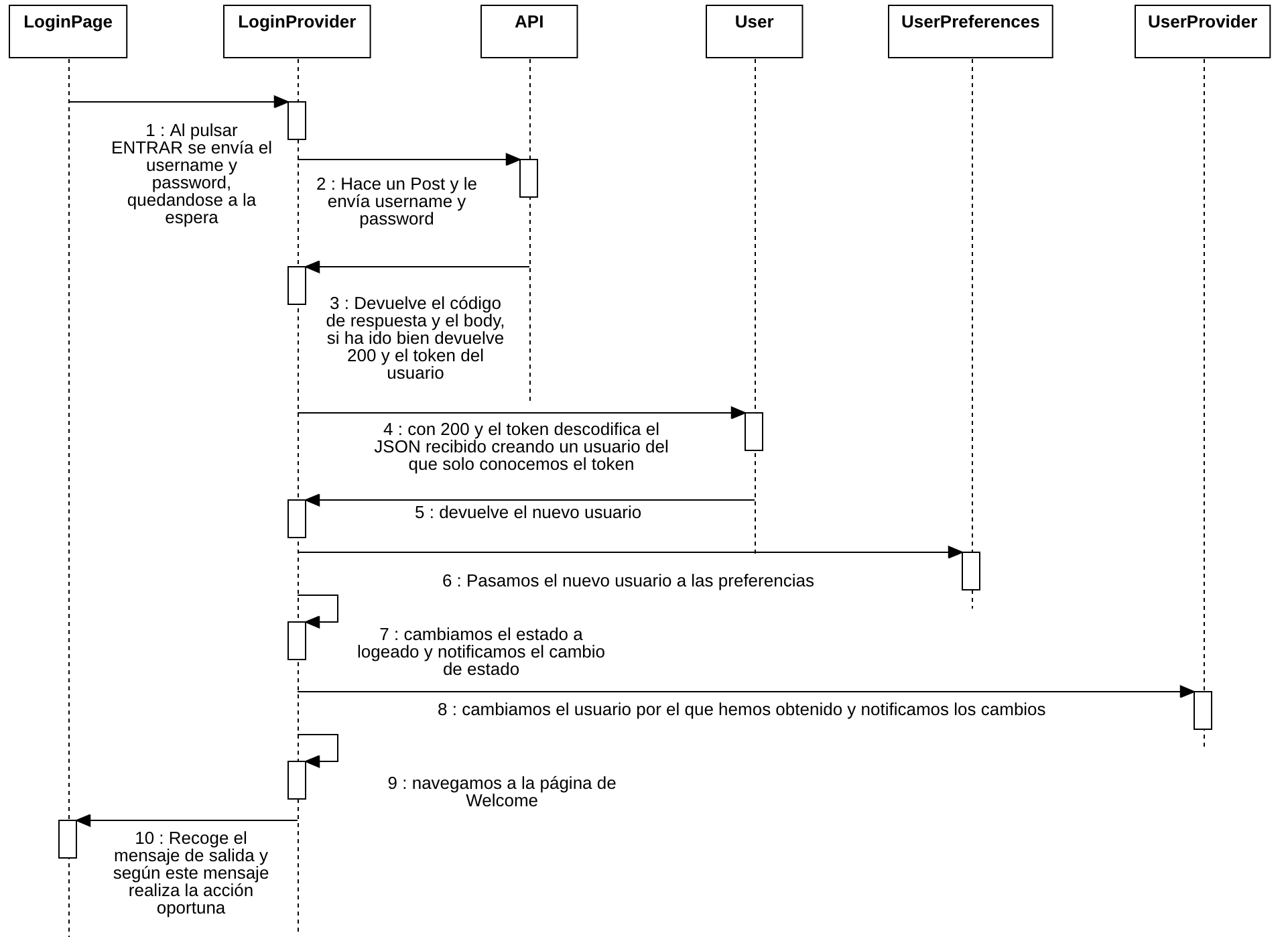
En el caso de poner un usuario correcto cambia a la página de Welcome, si las credenciales no son correctas muestra el mensaje 5 segundos y como no le hemos dado ruta se queda con el loader puesto.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamenteInterfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

### ESQUEMA PASOS LOGIN



## PÁGINA REGISTRO

Vamos a la carpeta **lib🡪pages** creamos una carpeta llamada **register** y dentro un archivo llamado *register\_page.dart.*

Creamos el esqueleto con un **statelessW.** Vamos al archivo *routes.dart* y añadimos la ruta de registro:

'register': (BuildContext context) => RegisterPage()

Volvemos a la página *register\_page.dart*

import 'package:delivery\_app/widgets/backButton.dart';

import 'package:flutter/material.dart';

class RegisterPage extends StatelessWidget {

TextEditingController emailCtrl = TextEditingController();

TextEditingController passCtrl = TextEditingController();

TextEditingController nameCtrl = TextEditingController();

TextEditingController apellidosCtrl = TextEditingController();

TextEditingController telefonoCtrl = TextEditingController();

@override

Widget build(BuildContext context) {

return Scaffold(

backgroundColor: Colors.white,

Cogemos el Appbar, le ponemos un color plano y le decimos que no queremos su flecha, queremos usar el backButton que programamos porque así podemos ponerle el color que queramos.

appBar: AppBar(

backgroundColor: Colors.white,

elevation: 0.0,

leading: Builder(

builder: (\_){

return backButton(Colors.black, context);

})

),

El **body** lo metemos dentro de widget *SingleChildScrollView* para que dependiendo de quien tenga el “focus” vaya haciendo el scroll.

body: SingleChildScrollView(

child: Center(

child: Container(

padding: EdgeInsets.all(30.0),

child: Column(

children: [

Text(

'Crear cuenta nueva',

style: TextStyle(

color: Theme.of(context).primaryColor,

fontWeight: FontWeight.bold,

fontSize: 30.0),

),

El *SizedBox* es como una caja “fantasma” que hace una separación entre el titulo y los botones.

SizedBox(height: 30.0,),

\_userNameInput(),

\_userApellidosInput(),

\_phoneInput(),

\_emailInput(),

\_passwordInput(),

\_registerButton(context),

Container(

padding: EdgeInsets.only(top:30.0),

child: Text('Presionando a registrar, acepta los siguientes Terminos y Condiciones' ,

textAlign: TextAlign.center,

style: TextStyle(

color: Colors.black,

fontWeight: FontWeight.w400,

fontSize: 13.0),

),

)

],

),

),

),

);

}

Para los inputs crearemos un Widget general. Dentro de la carpeta **widgets** creamos un input llamado *input.dart.* Vemos su implementación más abajo.

Widget \_userNameInput() {

return inputWidget(nameCtrl, 'Nombre', false, TextInputType.name);

}

Widget \_userApellidosInput() {

return inputWidget(apellidosCtrl, 'Apellidos', false, TextInputType.text);

}

Widget \_phoneInput() {

return inputWidget(telefonoCtrl, 'Telefono', false, TextInputType.phone);

}

Widget \_emailInput() {

return inputWidget(emailCtrl, 'Email', false, TextInputType.emailAddress);

}

Widget \_passwordInput() {

return inputWidget(passCtrl, 'Contraseña', true, TextInputType.text);

}

Widget \_registerButton(BuildContext context) {

return Container(

width: 370,

height: 45,

margin: EdgeInsets.only(top: 40),

child: ElevatedButton(

style: ElevatedButton.styleFrom(

primary: Theme.of(context).accentColor,

shape: RoundedRectangleBorder(

borderRadius: BorderRadius.circular(20.0)),

),

child: Text('Crear Cuenta'),

***\* Explicación lógica REGISTRO: En este punto tendremos que hacer el provider para acceder a la API y lo explicamos más adelante.***

onPressed: () => {

Aquí es donde vamos a hacer la lógica. Primero haremos un registro, el registro “atacará” a la tabla **users**, devolverá el *id* del usuario y en base a ese *id*, atacaremos a la tabla **cliente** y crearemos un cliente con ese *id* de usuario ya puesto. En realidad, atacaremos dos veces a la API. También podríamos modificar la API para que en una solo llamada recibierá toda la información.

Con la información que tenemos en los inputs crearemos un Map con la información del nuevo usuario.

Map<String, dynamic> newUser = {

'user': {

'email': emailCtrl.text,

'password': passCtrl.text,

},

'rol': 'ROLE\_CLIENTE'

};

Crearemos otro Map con los datos del usuario.

Map<String, dynamic> userData = {

'nombre': nameCtrl.text,

'apellidos': apellidosCtrl.text,

'telefono': telefonoCtrl.text,

};

Con esto llamamos al provider y le pasamos el nuevo usuario y los datos del usuario.

RegisterProvider provider =

Provider.of<RegisterProvider>(context, listen: false);

final Future<String> text = provider.register(newUser, userData, context);

Después escucharemos a nuestro provider, si la respuesta es “Registering” pondremos el loader que hicimos anteriormente y si no lo es cerraremos el loader con **Navigator.pop.**

provider.regStatus == RegStatus.Registering

? loading(context)

: Navigator.pop(context);

Cuando reciba el resultado del texto mostrará por pantalla el mensaje que corresponda.

text.then((value) {

if (provider.regStatus == RegStatus.NotRegistered) {

snackBar(context, value);

}

});

}),

);

}

}

Para acceder a la página de registro tenemos que ir a la página *login\_page.dart* y en la parte donde habíamos puesto el texto de ‘Registro’ lo envolvemos con un Widget (**Cmd+.** o **Ctl+.**) *GestureDetector* y podemos decir que gesto queremos detectar, en nuestro caso el click que sería el **onTap.**

GestureDetector(

onTap: () => Navigator.pushNamed(context, 'register'),

child: Text(

'Registro',

style: TextStyle(

color: Theme.of(context).accentColor,

fontWeight: FontWeight.w400,

fontSize: 15.0),

),

)

## PÁGINA INPUT

import 'package:flutter/material.dart';

Widget inputWidget(TextEditingController ctrl, String hint, bool obscure,

TextInputType keyboard) {

return Container(

margin: EdgeInsets.only(top: 10.0),

padding: EdgeInsets.only(left: 20.0),

decoration: BoxDecoration(

color: Color.fromRGBO(142, 142, 147, 1.2),

borderRadius: BorderRadius.circular(40.0)),

child: TextField(

controller: ctrl,

obscureText: obscure,

keyboardType: keyboard,

decoration: InputDecoration(

hintText: hint,

border: OutlineInputBorder(borderSide: BorderSide.none)),

),

);

}

## EXPLICACIÓN LÓGICA REGISTRO

\* ***Explicación lógica REGISTRO***: Como explicamos vamos a utilizar los providers específicos de las páginas para ponerlos dentro de la carpeta de esa página y los providers generales que “ataquen” al modelo de datos estarán en la carpeta **providers**.

Primero crearemos el provider propio y dentro de la carpeta **register** haremos una carpeta llamada **provider** y dentro un archivo llamado *register\_provider.dart*.

import 'dart:convert';

import 'package:flutter/material.dart';

import 'package:http/http.dart';

import '../../../../models/user.dart';

import '../../../api/api\_url.dart';

enum RegStatus { NotRegistered, Registered, Registering }

class RegisterProvider extends ChangeNotifier {

RegStatus \_regStatus = RegStatus.NotRegistered;

set regStatus(RegStatus status) {

\_regStatus = status;

}

RegStatus get regStatus => \_regStatus;

Hacemos un método asíncrono que hace la llamada a la API para crear un usuario.

Future<String> register(Map<String, dynamic> newUser,

Map<String, dynamic> userData, BuildContext context) async {

Ponemos el estado a registrando y notificamos el cambio.

\_regStatus = RegStatus.Registering;

notifyListeners();

El método esperará al resultado de la llamada a la API, en el que le envía el nuevo usuario (token e identificador)

Response response = await post(Uri.parse(Api.register),

body: json.encode(newUser),

headers: {

'Content-Type': 'application/json',

});

Si la respuesta es correcta devuelve el token y el identificador del usuario creado. Guarda en los datos del usuario el identificador obtenido y crea el nuevo cliente con los datos y el identificador, con el método siguiente *setUserData*.

if (response.statusCode == 200) {

User user = User.fromJson(response.body);

userData['user'] = user.id;

bool msg = await setUserData(userData);

if (msg) {

Navigator.pushReplacementNamed(context, 'login');

return '';

}

}

Si la respuesta es incorrecta devuelve un error. Lo más correcto sería devolver diferentes respuestas según el código de error obtenido.

\_regStatus = RegStatus.NotRegistered;

notifyListeners();

return 'Fallo de registro, intentelo de nuevo';

}

Hacemos otro método que será el que interactúe con la API para crear el usuario.

Future<bool> setUserData(Map<String, dynamic> userData) async {

Response response = await post(Uri.parse(Api.clienteCreate),

body: json.encode(userData),

headers: {

'Content-Type': 'application/json',

});

if (response.statusCode == 200) return true;

return false;

}

}

Tenemos que incluir el provider en el *main.dart*, en su array de **MultiProviders**. Básicamente lo que hacemos es envolver toda la aplicación con los *ChangeNotifier* y así puede acceder a todos los Widgets.

providers: [

ChangeNotifierProvider(create: (\_) => LoginProvider()),

ChangeNotifierProvider(create: (\_) => UserProvider()),

ChangeNotifierProvider(create: (\_) => RegisterProvider())

],

También tenemos que incluir en *api\_url.dart* la url de los enpoint a los que queremos acceder.

static const register = baseUrl + '/user/create';

static const clienteCreate = baseUrl + '/cliente/create'

## MODELO DE DATOS: CLIENTE

Tenemos que crear el modelo de datos cliente y para ello copiaremos el JSON desde Postman

{

"id": 1,

"nombre": "Pepe",

"apellidos": "Martinez",

"telefono": "678765456"

}

Y en Visual Studio Code pulsamos en **Ver🡪Paleta de Comandos** y elegimos la opción *JSON to Dart:from Clipboard.* Le ponemos de nombre UserData y a las preguntas wue nos va haciendo le decimos que no excepto a la última (*Implement Map/JSON method Code/Encode🡪 Yes)* y elegimos la carpeta **models** para guárdalo.

class UserData {

int? id;

String? nombre;

String? apellidos;

String? telefono;

UserData({this.id, this.nombre, this.apellidos, this.telefono});

factory UserData.fromJson(Map<String, dynamic> json) => UserData(

id: json['id'] as int?,

nombre: json['nombre'] as String?,

apellidos: json['apellidos'] as String?,

telefono: json['telefono'] as String?,

);

Map<String, dynamic> toJson() => {

'id': id,

'nombre': nombre,

'apellidos': apellidos,

'telefono': telefono,

};

}

Este sería el resultado, después de pulsar ENTRAR en la pantalla de bienvenida aparece la pantalla de LOGIN y su pulsamos sobre la palabra REGISTRO aparece la pantalla CREAR CUENTA, al rellenar los datos y pulsar CREAR CUENTA vuelve a la pantalla de LOGIN. Para completar el proceso faltaría la validación de datos para la creación de la cuenta.

|  |  |  |
| --- | --- | --- |
|  | Interfaz de usuario gráfica, Texto, Aplicación  Descripción generada automáticamente | Interfaz de usuario gráfica, Texto, Aplicación  Descripción generada automáticamente |

## MODIFICACIÓN RETORNO LOGIN RECIBE ID CLIENTE

Modificamos el modelo del *user.dart* y tenemos que añadirle el cliente.

import 'dart:convert';

class User {

int? id;

int? cliente;

String? email;

List<dynamic>? roles;

String? password;

String? token;

User({this.id, this.email, this.roles, this.password, this.token, this.cliente});

factory User.fromMap(Map<String, dynamic> data) => User(

id: data['id'] as int?,

cliente : data['cliente'] as int?,

email: data['email'] as String?,

roles: data['roles'] as List<dynamic>?,

password: data['password'] as String?,

token: data['token'] as String?);

Map<String, dynamic> toMap() => {

'id': id,

'cliente': cliente,

'email': email,

'roles': roles,

'password': password,

'token': token

};

/// `dart:convert`

///

/// Parses the string and returns the resulting Json object as [User].

factory User.fromJson(String data) {

return User.fromMap(json.decode(data) as Map<String, dynamic>);

}

/// `dart:convert`

///

/// Converts [User] to a JSON string.

String toJson() => json.encode(toMap());

}

Modificamos el archivo *shared\_preferences.dart* para que reciba el identificador del cliente

import 'package:shared\_preferences/shared\_preferences.dart';

import '../../models/user.dart';

class UserPreferences {

Future<bool> saveUserOnLogin(User user) async {

final SharedPreferences prefs = await SharedPreferences.getInstance();

prefs.setString('token', user.token!);

prefs.setInt('user\_id', user.id!);

prefs.setInt('id\_cliente', user.cliente!);

return true;

}

}

## MODELO DATOS: DIRECCIÓN

Vamos a POSTMAN y probamos la dirección de uno de los clientes, en la url le pasamos el id del cliente, en autentificación le damos su token y al enviar vemos todos los datos de las direcciones de ese cliente en la tabla.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Copiaremos el resultado del JSON del POSTMAN para crear el modelo de datos de direcciones, desde **Ver🡪Paleta de comandos** elegimos la opción *JSON to Dart: From Clipboard,* y hacemos los mismos pasos que hemos hecho en otras ocasiones.

Genera dos archivos, un *direcciones.dart* que es una lista de direcciones y *direccione.dart* que es la dirección en sí, le cambiamos el nombre *direccion.dart*.

import 'dart:convert';

import 'municipio.dart';

import 'provincia.dart';

class Direccion {

int? id;

String? calle;

String? numero;

String? puertaPisoEscalera;

int? codPostal;

Municipio? municipio;

Provincia? provincia;

Direccion({

this.id,

this.calle,

this.numero,

this.puertaPisoEscalera,

this.codPostal,

this.municipio,

this.provincia,

});

factory Direccion.fromMap(Map<String, dynamic> data) => Direccion(

id: data['id'] as int?,

calle: data['calle'] as String?,

numero: data['numero'] as String?,

puertaPisoEscalera: data['puertaPisoEscalera'] as String?,

codPostal: data['codPostal'] as int?,

municipio: data['municipio'] == null

? null

: Municipio.fromMap(data['municipio'] as Map<String, dynamic>),

provincia: data['provincia'] == null

? null

: Provincia.fromMap(data['provincia'] as Map<String, dynamic>),

);

Map<String, dynamic> toMap() => {

'id': id,

'calle': calle,

'numero': numero,

'puertaPisoEscalera': puertaPisoEscalera,

'codPostal': codPostal,

'municipio': municipio?.toMap(),

'provincia': provincia?.toMap(),

};

/// `dart:convert`

///

/// Parses the string and returns the resulting Json object as [Direccione].

factory Direccion.fromJson(String data) {

return Direccion.fromMap(json.decode(data) as Map<String, dynamic>);

}

/// `dart:convert`

///

/// Converts [Direccion] to a JSON string.

String toJson() => json.encode(toMap());

}

import 'dart:convert';

import 'direccion.dart';

class Direcciones {

List<Direccion>? direcciones;

Direcciones({this.direcciones});

factory Direcciones.fromMap(Map<String, dynamic> data) => Direcciones(

direcciones: (data['direcciones'] as List<dynamic>?)

?.map((e) => Direccion.fromMap(e as Map<String, dynamic>))

.toList(),

);

Map<String, dynamic> toMap() => {

'direcciones': direcciones?.map((e) => e.toMap()).toList(),

};

/// `dart:convert`

///

/// Parses the string and returns the resulting Json object as [Direcciones].

factory Direcciones.fromJson(String data) {

return Direcciones.fromMap(json.decode(data) as Map<String, dynamic>);

}

/// `dart:convert`

///

/// Converts [Direcciones] to a JSON string.

String toJson() => json.encode(toMap());

}

## PÁGINA COMENZAR PEDIDO

Dentro de la carpeta **lib🡪config🡪pages** creamos una carpeta llamada **comenzar\_pedido** y dentro de esta un archivo llamado *comenzar\_pedido\_page.dart.*

Creamos la página desde **statefulW.**

Reutilizaremos la página de *login\_page* cambiando alguna cosa.

import 'package:flutter/material.dart';

import 'package:flutter/services.dart';

import '../../colors/colors.dart';

class ComenzarPedido extends StatefulWidget {

ComenzarPedido({Key? key}) : super(key: key);

@override

State<ComenzarPedido> createState() => \_ComenzarPedidoState();

}

class \_ComenzarPedidoState extends State<ComenzarPedido> {

@override

Widget build(BuildContext context) {

var ancho = MediaQuery.of(context).size.width;

var alto = MediaQuery.of(context).size.height;

SystemChrome.setSystemUIOverlayStyle(SystemUiOverlayStyle.light.copyWith(

statusBarColor: Colors.transparent,

statusBarIconBrightness: Brightness.light,

));

return Scaffold(

body: Container(

child: SingleChildScrollView(

physics: AlwaysScrollableScrollPhysics(),

child: Column(

children: [

Stack(

children: [

Image(

width: double.infinity,

height: alto \* 0.5,

fit: BoxFit.cover,

image: AssetImage('assets/images/login-fondo.jpeg'),

),

],

),

Transform.translate(

offset: Offset(0.0, -20),

child: Container(

width: double.infinity,

height: alto \* 0.5,

decoration: BoxDecoration(

color: Colors.white,

borderRadius: BorderRadius.circular(20.0)),

child: Padding(

padding: EdgeInsets.all(20.0),

child: Column(

children: [

Text('Comienza tu pedido',

style: TextStyle(

color: Theme.of(context).primaryColor,

fontWeight: FontWeight.bold,

fontSize: 26.0,

)),

Text(

'Selecciona una dirección y el día y hora de reparto',

style: TextStyle(

color: gris,

fontWeight: FontWeight.w500,

fontSize: 13.0,

)),

Container(

margin: EdgeInsets.only(top: 20.0),

child: Align(

alignment: Alignment.centerRight,

child: GestureDetector(

onTap: () => {

//TODO 1: Abrir modal para dirección

},

child: Text(

'Crear nueva dirección',

style: TextStyle(

color: Colors.green,

fontWeight: FontWeight.w400,

fontSize: 12.0),

),

),

)),

Container(

margin: EdgeInsets.only(top: 20.0),

child: Row(

mainAxisAlignment: MainAxisAlignment.center,

children: [

Text(

'Dirección de envío',

style: TextStyle(

color: gris,

fontWeight: FontWeight.w500,

fontSize: 15.0),

),

],

),

),

//TODO 2: Consumer con o la direccion o un boton para seleccionar la direccion desde un modal

Container(

margin: EdgeInsets.only(top: 20.0),

child: Row(

mainAxisAlignment: MainAxisAlignment.center,

children: [

Text(

'Fecha de entrega',

style: TextStyle(

color: gris,

fontWeight: FontWeight.w500,

fontSize: 15.0),

),

],

),

)

//TODO 3: Consumer con la fecha o un boton para seleccionar la fecha

],

),

),

),

)

],

),

),

),

);

}

}

Actualmente se vería así:

Una captura de pantalla de un celular con texto e imagen

Descripción generada automáticamente con confianza media

Vamos a añadir la ruta.

final routes = <String, WidgetBuilder>{

'welcome': (BuildContext context) => WelcomePage(),

'login': (BuildContext context) => LoginPage(),

'register': (BuildContext context) => RegisterPage(),

'comenzar': (BuildContext context) => ComenzarPedido()

};

*Para ir comprobando y que no tengamos que hacer login cada vez en welcome.dart cambiamos la ruta al pulsar el botón ENTRAR para que vaya a “comenzar”.*

onPressed: () => Navigator.pushNamed(context, 'comenzar'),

## EXPLICACIÓN LÓGICA COMENZAR PEDIDO

Empezaremos por la dirección, para establecer la provincia haremos un desplegable donde salgan todas las provincias. Cuando seleccionemos una provincia aparecerá otro desplegable con los municipios de esa provincia. Tenemos que hacer dos llamadas a la API, una para coger la lista de provincias y otra para coger la lista de municipios en función de la provincia seleccionada.

### MODIFICAR USER PREFERENCES

Vamos a estar utilizando todo el tiempo el *token* y el *identificador del cliente*. Para no tener que cargarlo cada vez, vamos a crear unos métodos para que esta información se guarde en memoria de forma que si no existe en memoria lo cargará, pero si ya existe lo cogerá de aquí.

import 'package:shared\_preferences/shared\_preferences.dart';

import '../../models/user.dart';

class UserPreferences {

Creamos dos variables para almacenar el token y el id de cliente en memoria, los inicializamos a vacío y a 0 respectivamente.

String \_inMemoryToken = '';

int \_cliente = 0;

Hacemos un método asíncrono que devolverá el token. Si el token ya está en memoria lo devuelve automáticamente y si está vacío esperamos a obtenerlo, una vez lo tengamos lo devolvemos.

Future<String> get userAccessToken async {

if (\_inMemoryToken.isNotEmpty) {

return \_inMemoryToken;

}

\_inMemoryToken = await \_loadTokenFromShared();

return \_inMemoryToken;

}

Tenemos que utilizar una variable auxiliar para recoger el token para que no falle la aplicación si el token devuelve null.

Future<String> \_loadTokenFromShared() async {

final SharedPreferences prefs = await SharedPreferences.getInstance();

var accessToken = '';

final token = prefs.getString('token');

if (token != null) {

accessToken = token;

}

return accessToken;

}

Hacemos lo mismo con el cliente.

Future<int> get cliente async {

if (\_cliente != 0) {

return \_cliente;

}

\_cliente = await \_loadClienteFromShared();

return \_cliente;

}

Future<int> \_loadClienteFromShared() async {

final SharedPreferences prefs = await SharedPreferences.getInstance();

var accessCliente = 0;

final cliente\_id = prefs.getInt('id\_cliente');

if (cliente\_id != null) {

accessCliente = cliente\_id;

}

return accessCliente;

}

Future<bool> saveUserOnLogin(User user) async {

final SharedPreferences prefs = await SharedPreferences.getInstance();

prefs.setString('token', user.token!);

prefs.setInt('user\_id', user.id!);

prefs.setInt('id\_cliente', user.cliente!);

return true;

}

}

Si hiciéramos el *logout* tendríamos que hacer un método para borrar todo esto.

### PROVIDER DIRECCIÓN

Creamos una carpeta llamada **provider** dentro de **lib🡪pages🡪comenzar\_pedido**, y dentro un archivo llamado *direcciones\_provider.dart*.

import 'package:flutter/material.dart';

import '../../../../models/direcciones/municipio.dart';

import '../../../../models/direcciones/provincia.dart';

La clase extiende de *ChangeNotifier*.

class DireccionesProvider extends ChangeNotifier {

Necesitamos dos listas, una con las provincias y otro con los municipios.

List<Provincia> provinciasList = [];

List<Municipio> municipiosList = [];

Una variable para saber si está cargando información o no.

bool loading = false;

Haremos un método asincrono para obtener el listado de las provincias.

Future<void> getProvincias() async {

Comprobamos si la lista de provincias está vacía. La lista de provincias siempre va a ser la misma, por lo tanto, si esta vacía llamo a la API y me la traigo. Si no está vacía no hace falta que haga la llamada a la API porque ya la tengo.

if (provinciasList.isEmpty) {

loading = true;

notifyListeners();

Como ya nos habremos autentificado para la aplicación las llamadas a la API se harán utilizando ese token. Por lo tanto, lo tenemos que recuperar y pasarlo en la cabecera en la propiedad *Authorization*.

String token = await UserPreferences().userAccessToken;

Response response = await get(Uri.parse(Api.provincias),

headers: {

'Content-Type': 'application/json',

'Authorization': 'Bearer $token'

});

Si la respuesta a la API ha sido satisfactoria creamos una lista de tipo dinámica decodificando el JSON de la respuesta. Después rellenamos la lista de provincias transformando la lista anterior.

if (response.statusCode == 200) {

List<dynamic> jsonList = json.decode(response.body);

provinciasList = List<Provincia>.from(jsonList.map((e) => Provincia.fromMap(e)));

Como la lista ya está llena avisamos a quien la esté esperando y marcamos el *loading* a true.

notifyListeners();

loading = false;

}

}

}

}

Necesitamos incluir en *api\_url.dart* la url de los enpoint para acceder al listado de provincias y al listado de municipios según la provincia (por eso en municipio incluimos una ‘/’ detrás.

static const provincias = baseUrl + '/provincias';

static const municipios = baseUrl + '/municipios/';

También, tenemos que incluir el provider en el *main.dart*, en su array de **MultiProviders**.

ChangeNotifierProvider(create: (\_) => DireccionesProvider())

### MODIFICAR PÁGINA COMENZAR PEDIDO

Vamos a modificar la página *comenzar\_pedido\_page.*dart para que haga la llamada a la API para que traiga las provincias. Esta página es un **StatefulWidget**, y cada vez que cambiamos algo de la página cambia, de forma que si hacemos una llamada a la API se volverá a hacer cada vez que cambiemos algo. Hay una forma de evitar que esto ocurra, y que solo la llame la primera vez. Eso se hace con el método *initState*:

class \_ComenzarPedidoState extends State<ComenzarPedido> {

@override

void initState() {

// TODO: implement initState

super.initState();

DireccionesProvider provider =

Provider.of<DireccionesProvider>(context, listen: false);

provider.getProvincias();

}

@override

Widget build(BuildContext context) {

…

}

}

Donde dejamos la marca //TODO 1: Abrir modal para dirección vamos a hacer el modal.

// TODO 1: Abrir modal para dirección

showMaterialModalBottomSheet(

expand: true,

context: context,

Para acceder a este método tenemos que construir la clase AddDireccion que explicamos en otro punto.

builder: (context) => AddDireccion(),

);

### PÁGINA ADDDIRECCION

Para hacer esta parte crearemos dentro de la carpeta **comenzar\_pedido** una carpeta llamada **modales** y dentro de esta un archivo llamado *add\_direccion.dart*.

import 'package:flutter/cupertino.dart';

import 'package:flutter/material.dart';

import 'package:modal\_bottom\_sheet/modal\_bottom\_sheet.dart';

import '../../../../models/direcciones/municipio.dart';

import '../../../../models/direcciones/provincia.dart';

class AddDireccion extends StatefulWidget {

AddDireccion({Key? key}) : super(key: key);

@override

State<AddDireccion> createState() => \_AddDireccionState();

}

class \_AddDireccionState extends State<AddDireccion> {

TextEditingController calleCtrl = TextEditingController();

TextEditingController numeroCtrl = TextEditingController();

TextEditingController puertaCtrl = TextEditingController();

TextEditingController codPostalCtrl = TextEditingController();

Provincia? provincia;

Municipio? municipio;

@override

Widget build(BuildContext context) {

return CupertinoPageScaffold(

navigationBar: CupertinoNavigationBar(

leading: Container(),

middle: Text('NUEVA DIRECCIÓN'),

),

child: SafeArea(

child: Center(

child: Container(

padding: EdgeInsets.all(30.0),

child: Column(

children: [

Text(

'Complete el formulario:',

style: TextStyle(

color: Theme.of(context).accentColor,

fontWeight: FontWeight.bold,

fontSize: 20.0),

),

\_nombreCalle(),

Row(

children: [

Flexible(child: \_numero(),),

Flexible(flex: 2, child: \_puerta())

],

),

\_codPostal(),

//TODO 4: consumer para cargar lista de provincias y municipios

\_addbutton()

],

),

),

),

),

);

}

Widget \_nombreCalle() {

return inputWidget(

calleCtrl, 'Nombre de la calle', false, TextInputType.text);

}

Widget \_numero() {

return Container(

width: MediaQuery.of(context).size.width \* 0.3,

child: inputWidget(

numeroCtrl, 'Número de la calle', false, TextInputType.text),

);

}

Widget \_puerta() {

return inputWidget(

puertaCtrl, 'Puerta, piso, escalera', false, TextInputType.text);

}

Widget \_codPostal() {

return inputWidget(

codPostalCtrl, 'Código Postal', false, TextInputType.number);

}

Widget \_addbutton() {

return Container(

width: 350.0,

height: 50.0,

margin: EdgeInsets.only(top: 30.0),

child: ElevatedButton(

onPressed: () {},

style: ElevatedButton.styleFrom(

primary: Theme.of(context).accentColor,

shape:

RoundedRectangleBorder(borderRadius: BorderRadius.circular(20.0)),

),

child: Text('Guardar'),

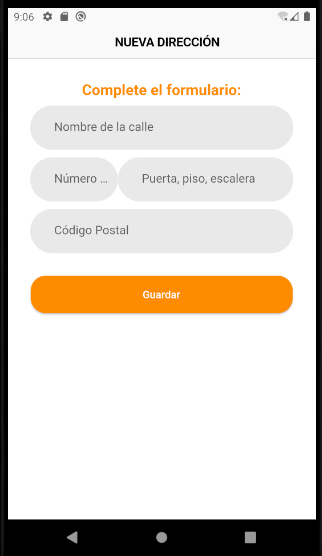
),

);

}

}

Actualmente se vería así:



Vamos a resolver el //TODO 4: consumer para cargar lista de provincias y municipios

Vamos a utilizar el **Consumer** que es otra forma de utilizar el provider. Lo bueno que tiene es que no hace falta instanciar el provider, pero hay que implementar más código. El **Consumer** es un Widget, que hay que decirle con qué tipo de provider va a trabajar. En el *builder* hay que ponerle una función que recibe 3 parámetros. En este caso el *child* es un Widget que no va a cambiar independientemente de lo que venga en el provider.

Consumer<DireccionesProvider>(

builder: (context, provider, child) {

Si la lista de provincias esta vacía devolvemos un texto de error.

if (provider.provinciasList.isEmpty) {

return Container(

child: Text(‘No hay provincias’),

);

}

Si la lista no está vacía vamos a una función que muestre el listado.

return \_dropdownProvinciaWidget(provider.provinciasList);

},

),

Consumer<DireccionesProvider>(

builder: (context, provider, child) {

if (provider.municipiosList.isEmpty) {

return Container(

child: Text('No hay municipios'),

);

}

return dropDownMunicipiosWidget(provider.municipiosList);

},

),

…

Widget \_dropdownProvinciaWidget(List<Provincia> provinciasList) {

return Container(

width: 350.0,

height: 50.0,

margin: EdgeInsets.only(top: 10),

padding: EdgeInsets.only(left: 20.0),

decoration: BoxDecoration(

color: Color.fromRGBO(142, 142, 147, 1.2),

borderRadius: BorderRadius.circular(40.0)),

Aquí va el listado de provincias. El child por defecto tiene una línea abajo, para esconderla envolvemos el select en un Widget que se llama *DropdownButtonHideUnderline.*

child: DropdownButtonHideUnderline(

Ahora hacemos un desplegable de objetos *Provincia*.

child: DropdownButton<Provincia>(

Ponemos un texto por defecto.

hint: Text('Seleccione provincia:'),

En valor tenemos que pasarle un valor externo por defecto para que se pinte de forma dinámica. Este valor tiene que ser nulo o tener un valor de los que luego tendrá la lista. Si no lo hacemos así luego nos dará error. En esta propiedad almacenaremos el valor seleccionado en la lista.

value: provincia,

Esta propiedad sirve para que se adapte al ancho del texto contenido.

isExpanded: true,

El método *onChanged* sirve para decir lo que va a pasar cuando seleccionemos algo de la lista.

onChanged: (Provincia? newProvincia) {

DireccionesProvider provider =

Provider.of<DireccionesProvider>(context, listen: false);

Falta hacer el *getMunicipios* que se traiga la lista de municipios en función del identificador de la provincia seleccionada (lo haremos en *DireccionesProvider*)

provider.getMunicipios(newProvincia?.id);

setState(() {

provincia = newProvincia;

municipio = null;

});

},

En *items* espera que le pasemos una lista desplegable cuyos elementos son Provincia.

items: provinciasList.map((Provincia value) {

return DropdownMenuItem<Provincia>(

value: value,

child: Text(value.provincia!)

);

}).toList(),

),

),

);

}

Widget dropDownMunicipiosWidget(List<Municipio> municipiosList) {

return Container(

width: 350.0,

height: 50.0,

margin: EdgeInsets.only(top: 10),

padding: EdgeInsets.only(left: 20.0),

decoration: BoxDecoration(

color: Color.fromRGBO(142, 142, 147, 1.2),

borderRadius: BorderRadius.circular(40.0)),

child: DropdownButtonHideUnderline(

child: DropdownButton<Municipio>(

hint: Text('Seleccione municipio:'),

value: municipio,

isExpanded: true,

onChanged: (Municipio? newMunicipio) {

setState(() {

municipio = newMunicipio;

});

},

items: municipiosList.map((Municipio value) {

return DropdownMenuItem<Municipio>(

value: value, child: Text(value.municipio!));

}).toList(),

),

),

);

}

### MODIFICAR PÁGINA DIRECCIONES PROVIDER

Añadimos el método *getMunicipio()*

Future<void> getMunicipios(int? id) async {

String token = await UserPreferences().userAccessToken;

if (id != null) {

Response response = await get(Uri.parse(Api.municipios + id.toString()),

headers: {

'Content-Type': 'application/json',

'Authorization': 'Bearer $token'

});

if (response.statusCode == 200) {

List<dynamic> jsonList = json.decode(response.body);

municipiosList =

List<Municipio>.from(jsonList.map((e) => Municipio.fromMap(e)));

notifyListeners();

}

}

}

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

## PARA PRÁCTICAR

Implementar el botón guardar para que se guarde la nueva dirección en el cliente.

Se podría hacer un **provider** con dos variables una que fuera la dirección seleccionada y la otra que fuera la fecha seleccionada. Al cargar esta pantalla veremos si está vacío, si no hay nada seleccionada de ninguna de los dos nos aparecerá un botón para añadir la fecha y la dirección, que será un modal en el que aparecerán todas las direcciones disponibles. El lugar para completarlo sería donde aparecen :

//TODO 2: Consumer con o la direccion o un boton para seleccionar la direccion desde un modal

…

//TODO 3: Consumer con la fecha o un boton para seleccionar la fecha