Flutter

Liqiang Yang - DAM2 - 2024-25

ÍNDICE

1. Estructura del proyecto:

1.1 Páginas

En total tenemos 6 páginas:

1.1.1 Main

¿Qué hace?

La funcionalidad de esta parte del código es inicializar la base de datos justo antes de iniciar la aplicación y llama a la página principal, Intropage.

1.1.2 Intropage

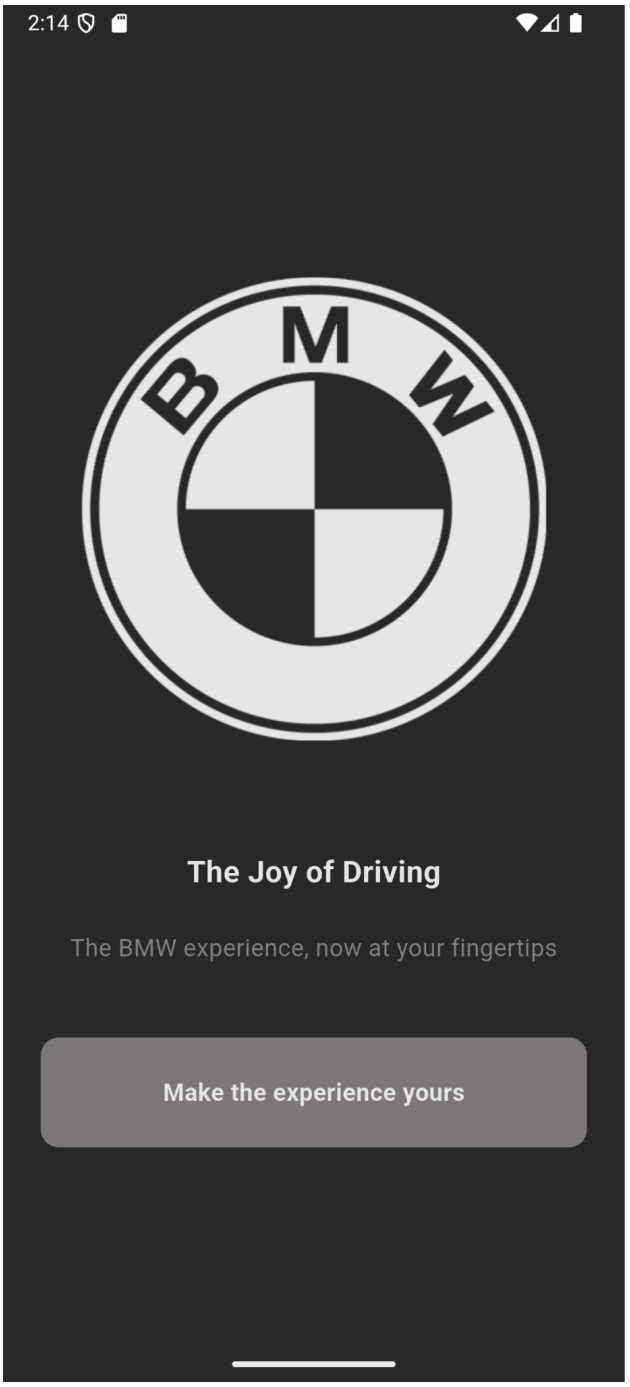
¿Qué hace?

Esta página no tiene ninguna función en específico, simplemente se usa para poder darle la bienvenida al usuario, para poder mostrar el logo y el eslogan.

¿Dependencias?

Antes de entrar en el asunto, en este proyecto se ha usado una paleta de colores negra y con tonos rojos, para ello hemos declarado la clase “darktheme” que se ha utilizado en todas las páginas para facilitar el cambio de colores sin tener que ir de un archivo a otro cambiando colores.

¿Diseño?



1.1.3 GNAV (Navigation Bar)

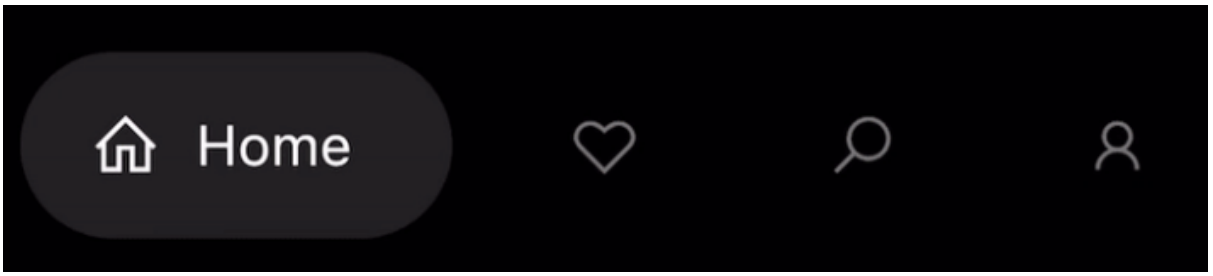
¿Qué hace?

Este archivo permite la navegación entre las tres pantallas principales de la app:

- Shop
- ShopCart
- Profile

¿Dependencias?

GNav, es una dependencia que se ha utilizado para crear la Bottom Navigation Bar, se tiene que importar y declarar en el “pubspec.yaml”.



Badges, es una dependencia que permite añadir insignias a cualquier cosa. Importante el uso de esta dependencia se tendrá que importar de la siguiente manera:

```
import 'package:badges/badges.dart' as <nombre>;
```

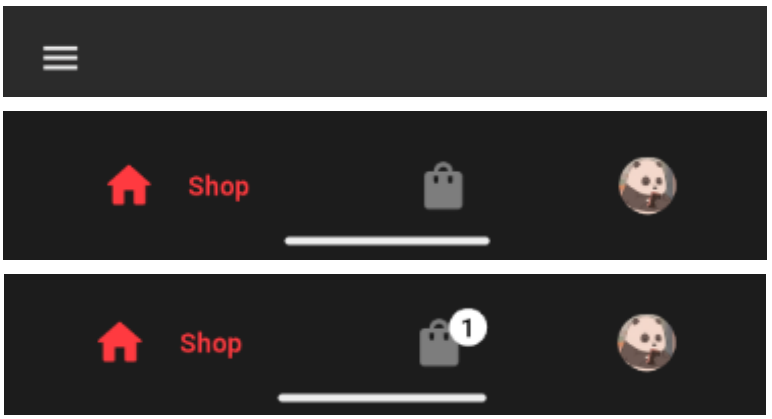
Donde nombre será cualquier cosa que queráis, porque Flutter de base nos ofrece Badges, eso se usará para diferenciar cuál utilizas.

¿Cómo funciona?

GNav, funciona de la misma forma que la Bottom Navigation Bar de Flutter por defecto, sin embargo, el GNav tiene una estética mucho más minimalista y elegante. Funciona de tal forma que después de la IntroPage se llama a “GNav” que tiene un Appbar y el propio Bottom Navigation bar, tenemos una lista de Widgets que se rellenan al utilizar el initState dandoles los valores de las 3 páginas principales, tiene una función llamada “*onTabChange*” que tiene un setState que cambia el body según el GButton apretado.

Además de ello, este tiene una función para poder actualizar el número de los Badges llamado “*updateBadgeCount*” que a la vez utiliza una query de la BBDD.

¿Diseño?



1.1.4 Shop

¿Qué hace?

Este archivo muestra los productos disponibles para el usuario, poder añadirlos al carrito, mediante un CustomScrollView se ha creado dos Scrolls, uno horizontal y otro vertical, se ha decidido usar este el CustomScrollView a un SingleScroll, por la personalización que ofrece el CustomScrollView y el objetivo principal era tener dos scrolls.

¿Clases / componentes?

ListProduct, clase que tiene listas de productos, en concreto dos:

- Wheels and Rims
- Models

Dichas listas se utilizan para poder darle un layout para poder generar los ItemTiles mediante ListView.
ItemTile, componente que se utiliza para poder dar diseño a los productos a exponer de las listas Wheels and Rims y la de Models.
BBDD, se utiliza para hacer inserts, updates…

¿Funciones y actualizaciones de las páginas?

Se ha mencionado durante la explicación de GNav, se usa una función *updateBudgetCount*, esta se pasa mediante el constructor del widget:

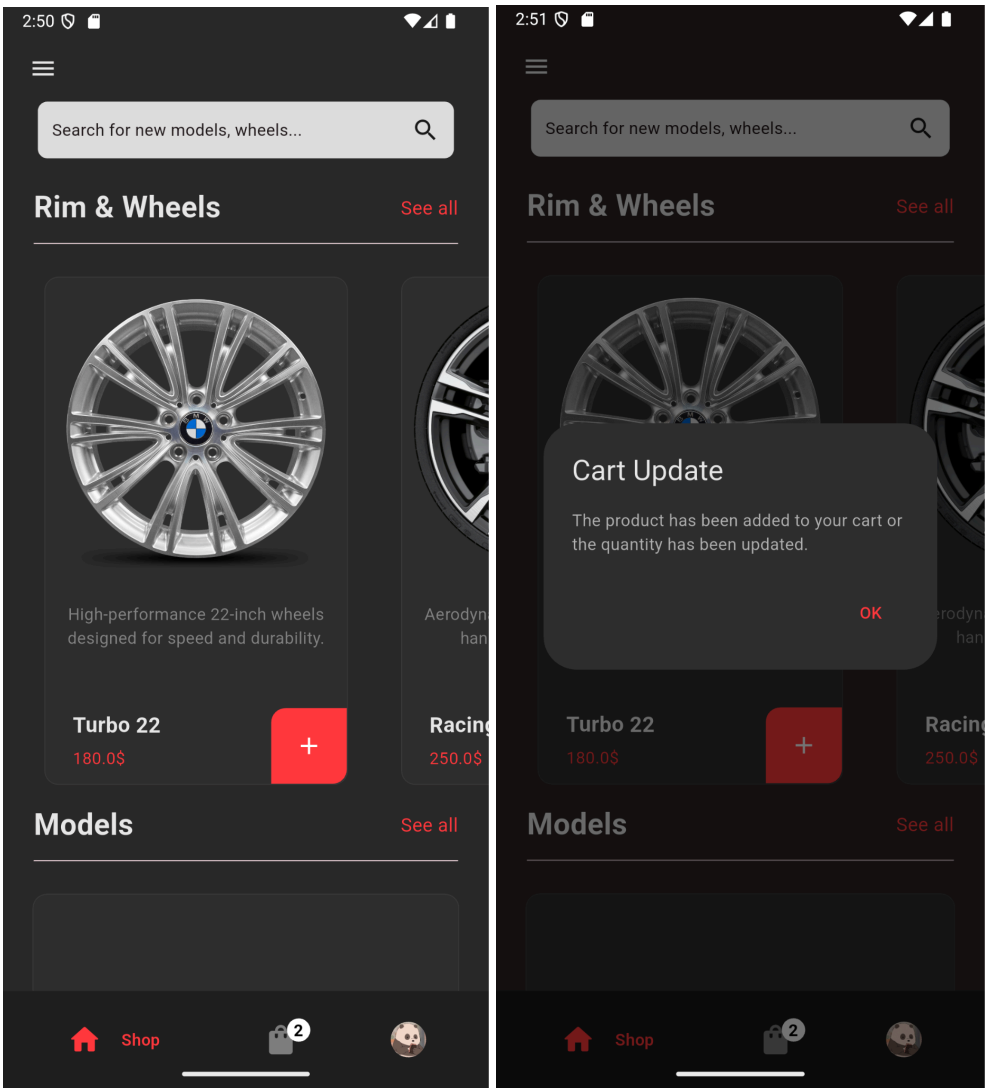
```
class Shop extends StatefulWidget {
  final VoidCallback onUpdate;
  const Shop({super.key, required this.onUpdate});

  @override
  State<Shop> createState() => _ShopState();
}
```

Como se puede observar arriba, se puede ver que tiene un VoidCallBack llamado "*onUpdate*", este se llamará cuando se añada un producto a la lista, es decir, cuando haga un insert a la Base de datos. Esto hará que cambie el número de la Navigation Bar.

¿Cómo se hace el insert o como se actualiza?
Este código se gestiona en el componente Item Tile. Pero haciendo un poco de spoiler se utiliza algo llamado "Raw query" dentro de la clase BBDD.

Además de ello, tiene una función llamada "*onProduct*" esta se llamaba cada vez que se añade un producto, la funcionalidad de esta es actualizar el badget del Navigation bar, y dar al usuario entender que ha agregado un producto o actualizado el carrito de compra, mediante un Alert Dialog.

¿Diseño?



1.1.5 ShopCart

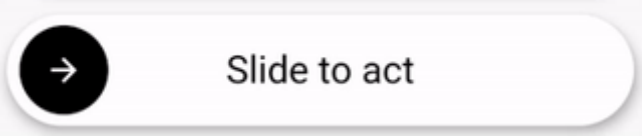
¿Qué hace?

Este archivo tiene el diseño del carrito de compras, detecta si hay artículos en el carrito, en caso de que no mostrará que no hay nada en la cesta, en caso de que sí mostrará los artículos en el carrito, mediante el uso del componente Cart Tile.

¿Clases / componentes / dependencias?

BBDD, se utiliza para hacer inserts, updates, delete, calcular el precio total de todos los artículos dentro del carrito.
Product, esta clase se utiliza para poder crear una lista de productos, la cual se usará para gestionar los selects de la base de datos, para así poder mostrarlos en la cesta, mediante un List View.
Cart Tile, componente que se utiliza para poder dar estilo a los artículos dentro de la cesta y tiene el control de poder aumentar, disminuir o eliminar el artículo directamente dentro del carrito de compras.

Slide to Act, dependencia usada para poder hacer el "*Slide to pay*" del apartado de checkout.



¿Funciones y actualizaciones de las páginas?

Similar a la página de Shop, esta también es necesario notificar al GNav de que ha cambiado las unidades dentro del carrito, porque dentro del carrito se puede eliminar, incrementar o disminuir la cantidad de artículos. Por ello, como hemos hecho anteriormente en Shop:

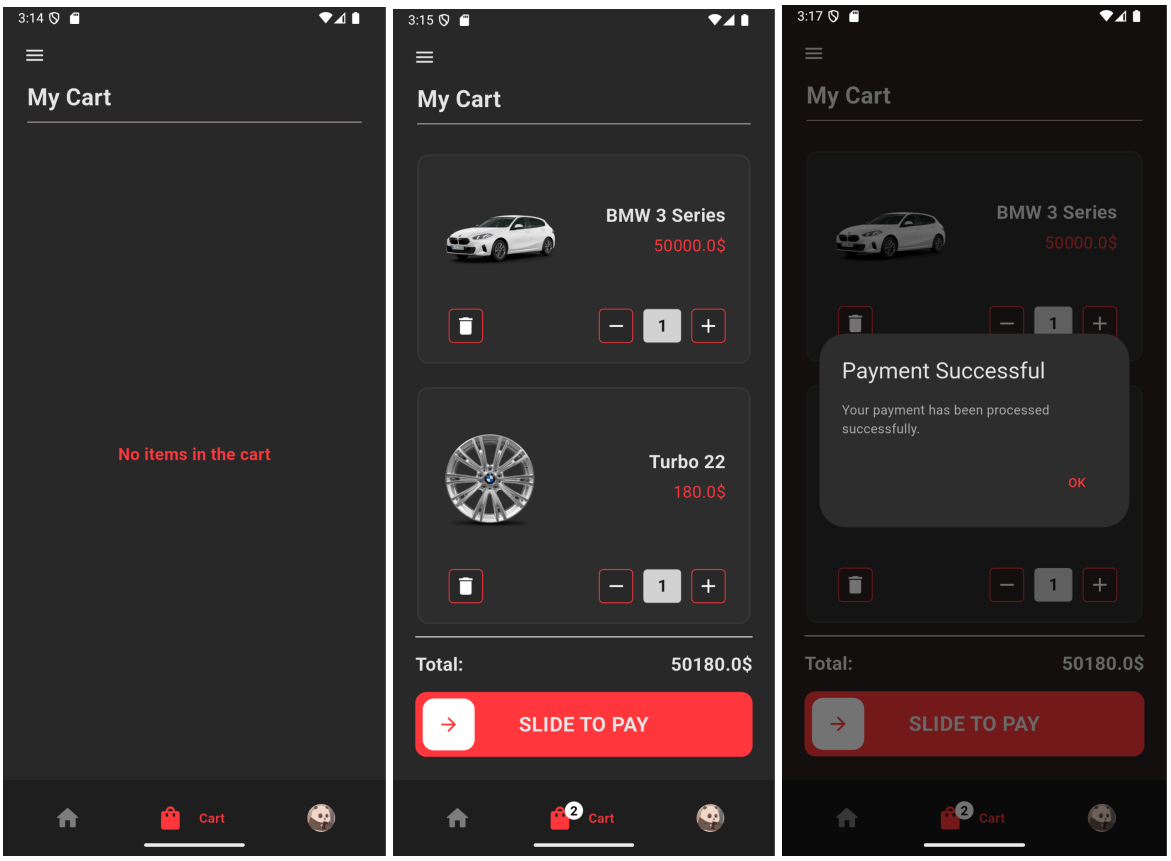
```
class Shopcart extends StatefulWidget {
  final VoidCallback onUpdate;
  const Shopcart({super.key, required this.onUpdate});

  @override
  State<Shopcart> createState() => _ShopcartState();
}
```

Como se puede observar arriba, se puede ver que tiene un VoidCallBack llamado "*onUpdate*", este se llamará cuando se añada un producto a la lista, es decir, cuando haga un insert, delete, update a la Base de datos. Esto hará que cambie el número de la Navigation Bar.

¡Importante!
En este archivo se gestiona, además de lo mencionado anteriormente, gestiona el precio total, añadir productos al carrito, cargar productos al carrito…

Para todas estas funciones se requiere hacer un setState "*loadCart*" esta función permite la actualización del badget y propio Widget ShopCart. Y todas estas funciones se han realizado con Querys a la base de datos.

¿Diseño?

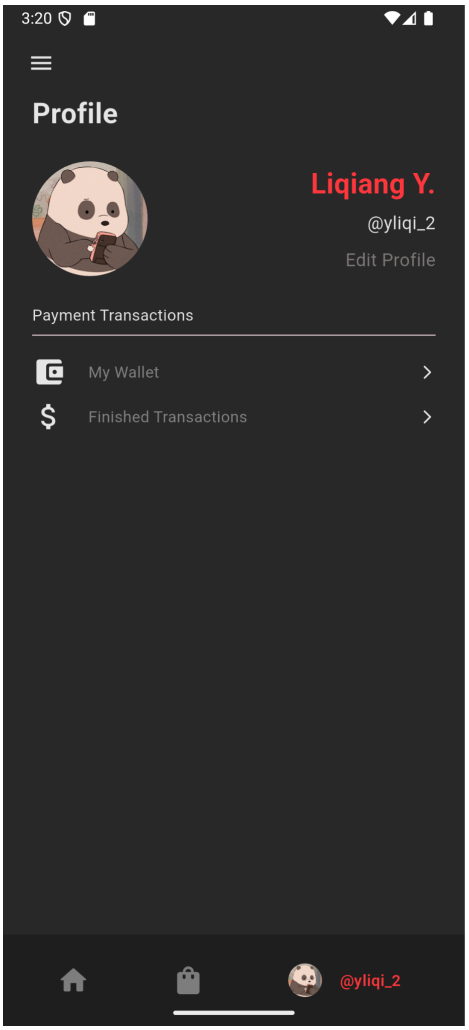


1.1.6 Profile

¿Qué hace?

Este archivo simplemente sirve para mostrar al usuario su perfil, se ha realizado este apartado pensando a futuro en caso de querer seguir ampliando la aplicación, implementarlo con Firebase, añadiendo la sección de pagos finalizados, etc.

¿Diseño?



1.2 Componentes

Para evitar tener código repetitivo se ha creado componentes, el diseño o plantilla reutilizable que pueda ser llamado para crear el producto deseado.

1.2.1 Item Tile

¿Qué hace?

Item Tile és el componente que permite al usuario añadir al carrito, haciendo un insert o update mediante la base de datos, es decir, gestiona los productos del carrito. Este se ha usado para la página de Shop, mediante un ListView se ha generado todos los productos disponibles a la venta.

¿Funciones o actualizaciones de las páginas?

Utiliza la función de "*insertOrUpdateProduct*" que gestiona el insert o update del producto a querer comprar.

Tiene un VoidCallBack para poder gestionar el badget y el alert dialog como hemos mencionado en Shop.

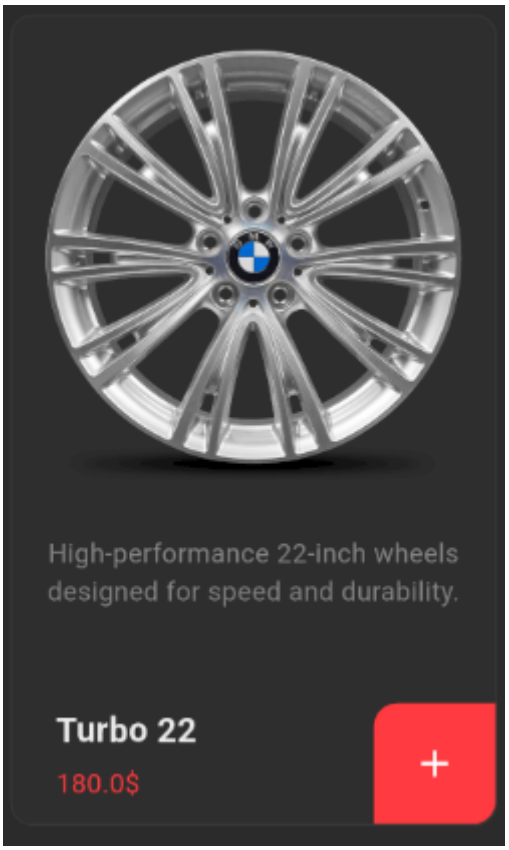
```
Widget itemTile(
    {required Product product,
    required double width,
    required VoidCallback onProduct})
```

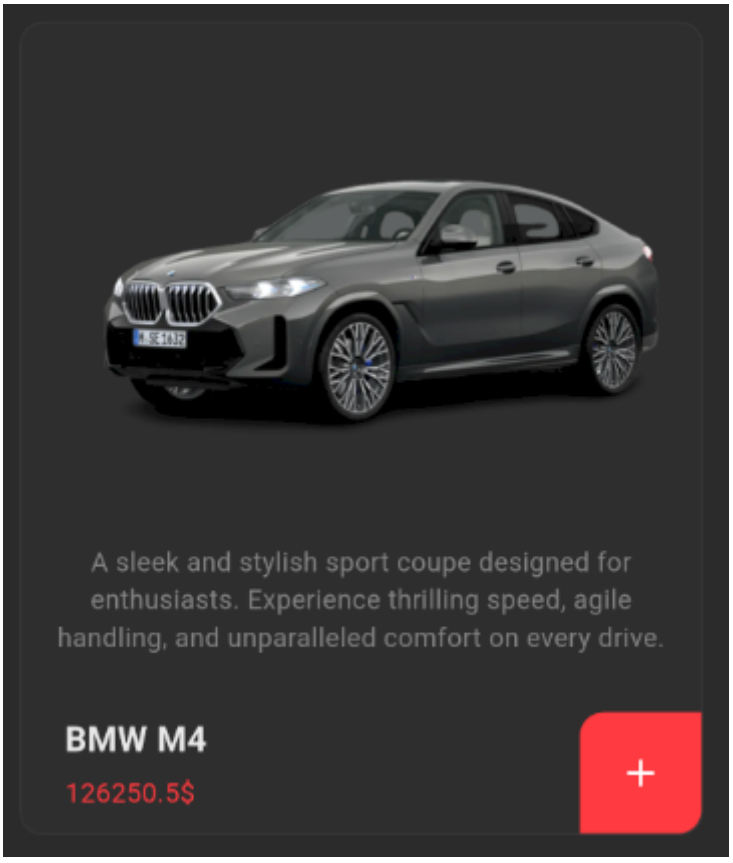
Como se puede observar también requiere un producto, es decir, a la hora de generar los widgets mediante el List View, y la lista de productos, se usa el índice para poder pasarle el producto, además del Width que es un MediaQuery para poder adaptarse a la pantalla de usuario para poder facilitar la visualización de los productos.

```
      ListView.builder(
                scrollDirection: Axis.horizontal,
                itemCount: productList.wheels.length,
                itemBuilder: (context, index) {
                  final product = productList.wheels[index];
                  return Padding(
                    padding: EdgeInsets.symmetric(
                        horizontal:
                             index == productList.wheels.length - 1 ? 20
: 10.0),
                    child: itemTile(
                      product: product,
                      width: widthItem,
                      onProduct: () => onProduct(context),
                    ),
                  );
                },
              ),
```

```
double screenWidth = MediaQuery.of(context).size.width;
    final widthItem = screenWidth * 0.625;
```

¿Diseño?


High-performance 22-inch wheels designed for speed and durability.
Turbo 22
180.0$
+


A sleek and stylish sport coupe designed for enthusiasts. Experience thrilling speed, agile handling, and unparalleled comfort on every drive.
BMW M4
126250.5$
+

1.2.2 Cart Tile

¿Qué hace?

Cart Tile es el componente que se muestra en la página de Shop Cart, además de tener los controles básicos del producto:

- +1 en el producto
- -1 en el producto
- Eliminar el producto del carrito

¿Funciones o actualizaciones de las páginas?

Como en el Item Tile, este tiene la misma mecánica para crearse, mediante una Lista de productos y el índice del ListView.

Para poder gestionar el producto tiene funciones como: (BBDD)

- decreaseOrAugmentProduct
- deleteProduct

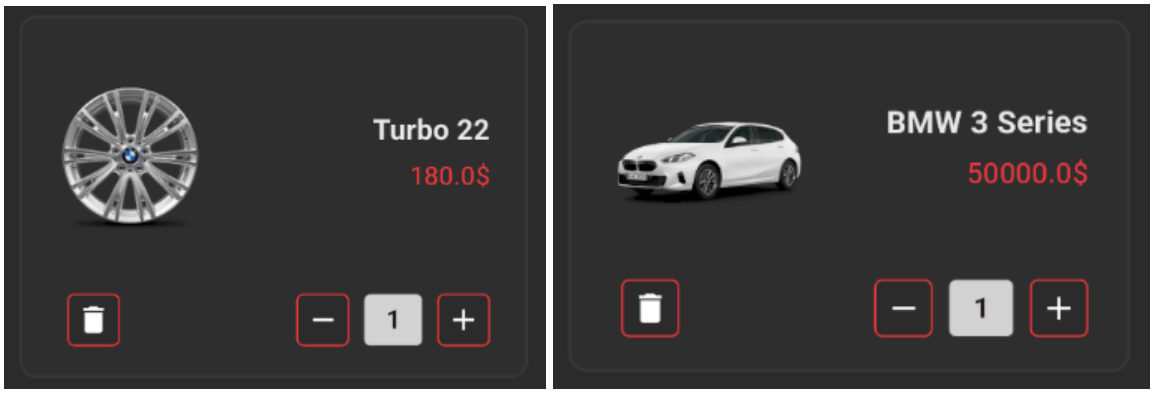
Para poder gestionar el Badget se utiliza:

- onUpdate (del constructor)

```
Widget cartTile(
    {required Product product,
    required VoidCallback onUpdate,
    required double width,
    required double height})
```

El width y el height, son MediaQuery realizados en el padre Shop Cart, utilizados para que sea responsivo en diferentes versiones de móvil, utilizadas para adaptar las fotos de los modelos o ruedas o llantas.

¿Diseño?



1.3 Clases

1.3.1 Base de datos

¿Qué hace?

Esta clase se encarga de gestionar todas las funciones relacionadas con el paquete SQFLITE.

¿Qué es SQFLITE y el uso de Path?

El SQFlite es un plugin que se usa para mantener una base de datos localmente para aplicaciones móviles. Este te permite la creación de tablas, hacer inserts, updates, deletes, todo lo básico de una base de datos.

El uso de Path, es debido a que este permite la adaptación de diferentes plataformas como Android o IOS…

¿Funciones?

Iniciar la Base de datos + creación de la tabla

Código de la clase BBDD:

```
Future<Database> getDatabase() async {
  return openDatabase(
    join(await getDatabasesPath(), 'product_db'),
    onCreate: (db, version) {
      return db.execute(
          'CREATE TABLE product(id INTEGER PRIMARY KEY, name TEXT,
price REAL, imgpath TEXT, desc TEXT, quantity INTEGER)');
    },
    version: 1,
  );
}
```

```
Future<void> initializeDatabase() async {
  await getDatabase();
}
```

Código del Main:

```
void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await initializeDatabase();
  runApp(const MyApp());
}
```

En el Main, nos aseguramos que el Widget se haya creado, en este caso el IntroPage, una vez asegurado se inicializa la base de datos y se crea la tabla que se va a usar, utilizando el join de la dependencia Path.

Insert or Update de productos

```
Future<void> insertOrUpdateProduct(Product product) async {
  final db = await getDatabase();

  await db.rawInsert('''
    INSERT INTO product (id, name, price, imgpath, desc, quantity)
    VALUES (?, ?, ?, ?, ?, 1)
    ON CONFLICT(id) DO UPDATE SET
      name = excluded.name,
      price = excluded.price,
      imgpath = excluded.imgpath,
      desc = excluded.desc,
      quantity = product.quantity + 1
  ''', [
    product.id,
    product.name,
    product.price,
    product.imgpath,
    product.desc,
  ]);
}
```

Como hemos mencionado anteriormente, esta es la función que se usa para manejar los inserts o updates dentro de la página Shop, esta función se aprovecha de que SQFLite tiene una opción para manejar los conflictos, es decir, si en el apartado de Shop, se aprieta muchas veces el “+”, la propia base de datos detectará el intento del insert de un producto existente, por tanto, se hará la suma en cantidad como se muestra en el código. En caso de que no exista, ignorara el código de abajo (ON CONFLICT), y se insertará con la información que le pasemos, excepto por la cantidad que por defecto se inserta con un 1.

Select de los productos en el carrito

```
Future<List<Product>> selectProductos() async {
  final db = await getDatabase();

  List<Map<String, Object?>> productMap = await db.query('product');

  return [
    for (final {
          'id': id as int,
          'name': name as String,
          'price': price as double,
          'imgpath': imgpath as String,
          'desc': desc as String,
          'quantity': quantity as int,
        } in productMap)
      Product(
        id: id,
        name: name,
        price: price,
        imgpath: imgpath,
        desc: desc,
        quantity: quantity,
      ),
  ];
}
```

Esta función devuelve todos los productos que hay en la tabla "*product*", en forma de lista.

Delete del producto con condición

```
Future<void> deleteProduct(int id) async {
  final db = await getDatabase();
  await db.delete(
    'product',
    where: 'id = ?',
    whereArgs: [id],
  );
}
```

Esta función, como se puede observar en el código, realiza un delete pasandole un ID, este ID será el product.id.

¡Importante!
whereArgs: [id], → Esta línea de aquí es una medida de seguridad que bloquea una inyección de sql.

Delete de todos los productos

```
Future<void> deleteAllProduct() async {
  final db = await getDatabase();
  await db.delete(
    'product',
  );
}
```

Función utilizada para “checkout”, elimina todos los productos que hay en el carrito, fingiendo la compra de estos.

Update del producto, increase or decrease

```dart
Future<void> decreaseOrAugmentProduct(int id, int action) async {
  final db = await getDatabase();

  final List<Map<String, Object?>> result = await db.query(
    'product',
    columns: ['quantity'],
    where: 'id = ?',
    whereArgs: [id],
  );

  if (result.isNotEmpty) {
    int valor = result.first['quantity'] as int;

    if (action == 0) {
      if (valor > 1) {
        await db.update(
          'product',
          {'quantity': valor - 1},
          where: 'id = ?',
          whereArgs: [id],
        );
      } else {
        await deleteProduct(id);
      }
    } else {
      await db.update(
        'product',
        {'quantity': valor + 1},
        where: 'id = ?',
        whereArgs: [id],
      );
    }
  }
}
```

Primero se realiza un select del producto que quieres aumentar o disminuir la cantidad, y se guarda en result, se hará una comprobación si result tiene valor, en caso de que tenga, entra en acción la segunda variable "*action*", este decide que botón fue presionado en caso del + este entrará al update sumando uno. En caso de que action sea 0, comprobará si el valor no sea 1, en caso de que sea 1, hará un delete, en caso de que tenga más de 1, se hara un update restandole la cantidad -1.

Select de la cantidad total de productos en la cesta

```
Future<int> getTotalQuantity() async {
  final db = await getDatabase();

  final List<Map<String, Object?>> result = await db.rawQuery('''
    SELECT SUM(quantity) as totalQuantity FROM product
  ''');

  if (result.isNotEmpty && result.first['totalQuantity'] != null) {
    return result.first['totalQuantity'] as int;
  }

  return 0;
}
```

Función que devuelve un int, la cantidad total de productos en el carrito, este se utiliza en el GNav, para poder actualizar el Badget.

Select del precio total de todos los productos

```
Future<double> getTotalPrice() async {
  final db = await getDatabase();

  final List<Map<String, Object?>> result = await db.rawQuery('''
    SELECT SUM(price*quantity) AS totalPrice FROM product
    ''');

  if (result.isNotEmpty && result.first['totalPrice'] != null) {
    return result.first['totalPrice'] as double;
  }

  return 0.0;
}
```

Función que devuelve un double, precio total de todos los productos sumados. Utilizado en la página de ShopCart.

1.3.2 List of Products

¿Qué hace?

Mediante la clase productos, se crea la lista de productos, con dos listas:

- Models
- Wheels and Rims

Se utiliza para poder generar en la página de Shop.

1.3.3 Products

¿Qué hace?

Declara la clase de productos, el constructor y la función que se necesita para la base de datos.

```
class Product {
  int id;
  String name;
  double price;
  String imgpath;
  String desc;
  int quantity;

  Product(
      {required this.id,
      required this.name,
      required this.price,
      required this.imgpath,
      required this.desc,
      required this.quantity});

  Map<String, Object?> toMap() {
    return {
      'id': id,
      'name': name,
      'price': price,
      'imgpath': imgpath,
      'desc': desc,
      'quantity': quantity
    };
  }
}
```

1.4 Constantes

1.4.1 DarkTheme

¿Qué hace?

Clase para poder organizar mejor el proyecto, que sea más modular, permite la expansión de dicho proyecto con más facilidad. Contiene los colores utilizados dentro del proyecto.

1.5 Pubspec.yaml

Se ha importado los assets para las imágenes usadas en dicha aplicación, además de todas las dependencias usadas en la aplicación.

2. GitHub

https://github.com/yliqi2/myshop