MonsterChef (Lista de listas)

15/05/2025

Representante: Agustín Ramos Torres, Correo: art00051@red.ujaen.es

Integrante: Francisco Jesús Senán Partal, Correo: fjsp0013@red.ujaen.es

Equipo 13

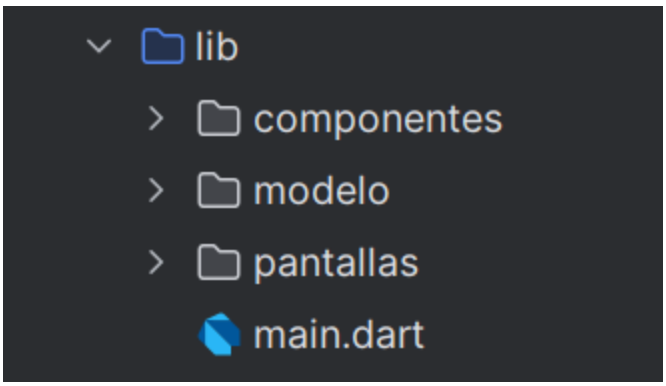
Enlace repositorio: https://github.com/AgustinRamosTorres/proyectoFinalMoviles

Desacoplamiento del código y patrones de diseño

Para realizar nuestra aplicación, hemos optado por realizar diversas configuraciones:

Patrón modelo - vista - controlador

Dicho patrón implica la división en módulos, en nuestro caso, tenemos los siguientes:



- **Componentes**: Encapsulamos la implementación de cada uno de los componentes de las diferentes partes, como por ejemplo, cada una de las líneas de los productos de las listas. (Controlador)
- **Modelo**: Encapsula la lógica de la aplicación, como por ejemplo, qué información tiene cada producto, que información tiene cada lista... (Modelo)
- **Pantallas**: Encapsula la interfaz de la aplicación, como las pantallas en las que se muestra la lista de productos. (Vista)

Dicho patrón favorece la limpieza y encapsulación de código, permitiendo el desacoplamiento y la integración de las clases de manera independiente.

Patrón singleton

Este patrón se caracteriza por la implementación de una única instancia del objeto al que se le aplique. En nuestro caso, tenemos dos objetos de los cuales buscamos tener únicamente una instancia, que son Lista de listas y Lista de productos. A las cuales accederemos desde diversos lugares de la aplicación, pero únicamente dispondremos de una sola instancia.

Patrón estado / Patrón observador

Estos patrones se caracterizan por proporcionar la comunicación del cambio a un grupo de integrantes (consumer) de los cambios producidos (provider).

Para implementar dicho patrón usaremos la metodología provider - consumer, que permite notificar a todas las pantallas cuyo estado es mutable, el cambio para que puedan volver a renderizar el widget.

Provider

```
IconButton(
  icon: const Icon(Icons.shopping_cart_checkout),
  onPressed: () {
    // 8
    final manager = Provider.of<ListaCompra>(context, listen: false);
    manager.borrarComprados(_idLista);
  },
),
```

Consumer

```
return Consumer<ListaListas>(
  builder: (context, manager, child) {
    if (manager.listas.isNotEmpty) {
      // 11
      /// Ojo cuidao
      return ListaListasPantallaLlena(listaListas: manager);
    } else {
      return const ListaListasPantallaVacia();
    }
  },
);
```

Todas estas configuraciones, nos han permitido realizar un desacoplamiento del código, realizando una implementación encapsulada cumpliendo uno de los principios SOLID (Single responsibility principle), caracterizado por independizar la modificación de la funcionalidad en componentes independientes y dividir las responsabilidades en los diferentes componentes.

Explicación de los requisitos (con capturas)

Gestión de almacenamiento de listas y de productos

Cada vez que se ejecute alguna modificación sobre la lista de listas de la compra, y sobre la lista de la compra, se realizará una escritura en memoria, para evitar las inconsistencias en el caso en el que se cierre la aplicación momentáneamente.

```
Future<void> _salvarListas() async {
  final file = await _localFile;

  var cadena = '[\n';
  for (var i = 0; i < _listas.length; i++) {
    cadena += _listas[i].aJson();
    if (i < _listas.length - 1) {
      cadena += ',\n';
    }else {
      cadena += '\n';
    }
  }
  cadena += ']';
  file.writeAsString(cadena);
}

Future<void> _leerListas() async {
  try{
    final file = await _localFile;
    final productosString = await file.readAsString();
    final List<dynamic> productosJson = jsonDecode(productosString);
    for (var prodJson in productosJson){
      _listas.add(Lista.desdeJson(prodJson));
    }
    notifyListeners();
  }on FileSystemException {
    return;
  }
}
```

```
Future<void> _salvarProductos() async {
  final file = await _localFile;

  var cadena = '[\n';
  for (var i = 0; i < _productos.length; i++) {
    cadena += _productos[i].aJson();
    if (i < _productos.length - 1) {
      cadena += ',\n';
    }else {
      cadena += '\n';
    }
  }
  cadena += ']';
  file.writeAsString(cadena);
}

Future<void> _leerProductos() async {
  try{
    final file = await _localFile;
    final productosString = await file.readAsString();
    final List<dynamic> productosJson = jsonDecode(productosString);
    for (var prodJson in productosJson){
      _productos.add(Producto.desdeJson(prodJson));
    }
    notifyListeners();
  }on FileSystemException {
    return;
  }
}
```

En ambos casos estamos implementando una función para salvar los cambios realizados en un archivo previamente creado, almacenado en forma de JSON, de ahí que nosotros, agreguemos como cadena de texto, mediante la función aJson.

Posteriormente cada vez que nosotros arranquemos la aplicación se llamarán a los métodos para leer desde esos archivos creados, los cuales, al guardarse en formato JSON, sería necesario volverlos a leer en formato JSON, de ahí que usemos el método desdeJson.

Añadir, cambiar nombre y borrar listas de la compra y productos

Atendiendo a los patrones de diseño comentados anteriormente, podemos ver aquí una referencia al patrón estado, mediante el notifyListeners, que permite avisar a los consumidores del recurso único, de los cambios realizados.

En los siguientes códigos, se vé como se puede borrar, mediante los métodos predefinidos del lenguaje para acceder a las estructuras necesarias.

```
void borraProducto(int indice) {
  _productos.removeAt(indice);
  _salvarProductos();
  notifyListeners();
}

void anadeProducto(Producto item) {
  _productos.add(item);
  _salvarProductos();
  notifyListeners();
}

void actualizaProducto(int indice, Producto item) {
  _productos[indice] = item;
  _salvarProductos();
  notifyListeners();
}
```

```
void borraLista(int indice) {
  _listas.removeAt(indice);
  _salvarListas();
  notifyListeners();
}

void anadeLista(Lista item) {
  _listas.add(item);
  _salvarListas();
  notifyListeners();
}

void actualizaLista(int indice, Lista item) {
  _listas[indice] = item;
  _salvarListas();
  notifyListeners();
}
```

Cambiar entre listas de forma sencilla

En Flutter, cuando utilizamos la funcionalidad NavigatorPush, nos permite realizar una interacción a 1 nivel, con la posibilidad de volver hacia atrás con una flecha que nos proporciona el propio componente. Por lo que desde listaListasPantallaLlena, mediante la función onTap al pulsar en el componente, podemos interactuar y volver hacia atrás mediante dicha flechita.

```
onTap: () {
  final listaCompra = Provider.of<ListaCompra>(
    context,
    listen: false,
  );
  Navigator.push(
    context,
    MaterialPageRoute(
      builder:
          (context) => ChangeNotifierProvider.value(
            value: listaCompra,
            child: ListaCompraPantalla(idLista: lista.id,),
          ),  // ChangeNotifierProvider.value
    ),  // MaterialPageRoute
  );
},
```


Productos
No hay productos todavia
Pulsa el boton + para añadirlos

Borrar elementos de la lista de la compra, deslizando hacia la izquierda (Además hemos hecho con las listas)

Para la implementación de dicha funcionalidad hemos utilizado el widget Dismissible, que permite realizar una acción en la dirección en la que le especifiquemos, en este caso hacia la izquierda, sería end To Start, y pondremos la acción a realizar en la función onDismissed.

Dicha funcionalidad se pedía para productos, pero se ha realizado, además para listas.

```
return Dismissible(
  key: Key(producto.id),
  direction: DismissDirection.endToStart,
  background: Container(
    color: Colors.red,
    alignment: Alignment.centerRight,
    child: const Icon(Icons.delete, color: Colors.white, size: 35),
  ),  // Container
  onDismissed: (direction) {
    listaCompra.borraProducto(index);
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(content: Text('${producto.nombre} eliminado')),
    );
  },
```

Permitir marcar con facilidad aquellos productos comprados

En este caso, en la implementación de cada una de las líneas de los productos, se ha implementado un CheckBox, que permite modificar el estado de cada producto, el cual se modifica con la función onChange, que se le pasará por cabecera, favoreciendo el desacoplamiento.

En este caso, la lógica que realiza sería tachar el producto para indicar que se ha comprado.

```
Row(
  children: <Widget>[
    Text(producto.cantidad.toString(), style: titleLarge),
    // 21
    Checkbox(value: producto.completado, onChanged: completar),
  ],  // <Widget>[]
),  // Row
```

Permitir borrar con facilidad todos los productos de una lista ya marcados como comprados

Para permitir dicha funcionalidad, en la clase ListaCompraPantalla, en la parte superior, se ha implementado un icono, el cual permite indicar de manera intuitiva, cuál es el botón para eliminar los ya comprados, el cual la funcionalidad se encapsula en un provider (Patrón estado), que elimina los productos cuyo estado está completado.

```
IconButton(
  icon: const Icon(Icons.shopping_cart_checkout),
  onPressed: () {
    // 8
    final manager = Provider.of<ListaCompra>(context, listen: false);
    manager.borrarComprados(_idLista);
  },
),  // IconButton
```

```
void borrarComprados(String idLista) {
  List<int> indices = [];
  for (var i = 0; i < _productos.length; i++) {
    if (_productos[i].listaId == idLista) {
      indices.add(i);
    }
  }

  for (int i = indices.length - 1; i >= 0; i--) {
    if (_productos[indices[i]].completado == true) {
      _productos.removeAt(indices[i]);
    }
  }

  _salvarProductos();
  notifyListeners();
}
```

Permitir borrar con facilidad todos los productos de una lista

Para permitir dicha funcionalidad, en la clase ListaCompraPantalla, en la parte superior, se ha implementado un icono, el cual permite indicar de manera intuitiva, cuál es el botón para eliminar todos los productos, el cual la funcionalidad se encapsula en un provider (Patrón estado), que elimina los productos disponibles en dicha lista.

```
IconButton(
  icon: const Icon(Icons.delete_forever),
  onPressed: () {
    // 8
    final manager = Provider.of<ListaCompra>(context, listen: false);
    manager.borraListaCompra(_idLista);
  },
),  // IconButton
```

Mostrar cada lista de forma que los productos ya marcados como comprados sigan siendo visibles, pero sin estar en medio

Para implementar dicha funcionalidad, como he comentado anteriormente, a el Checkbox se le proporciona una función anónima que se le pasa por cabecera para favorecer el desacoplamiento. Dicha función debe contener una llamada a este método, el cual, cuando se llama, se cambia su estado a completado, además de cambiar la posición en la lista, para que baje hacia la parte inferior de esta, además cuando se desmarque, está pasará a la parte superior de la lista para una correcta visualización e interacción amigable con el usuario.

```
void marcaCompletado(String idLista, int indice, bool completado) {
  List<int> indices = [];
  for (var i = 0; i < _productos.length; i++) {
    if (_productos[i].listaId == idLista) {
      indices.add(i);
    }
  }

  final producto = _productos[indices[indice]];

  if (completado == true){
    _productos.removeAt(indices[indice]);
    _productos.add(producto.copiaSinNulo(completado: completado));
  }else {
    _productos.removeAt(indices[indice]);
    _productos.insert(0,producto.copiaSinNulo(completado: completado));
  }

  _salvarProductos();
  notifyListeners();
}
```

(Extra +0,75) Permitir la realización de otra operación (distinta del borrado de productos y el marcado de un producto como prioritario) al deslizar hacia la derecha sobre un producto

Para implementar dicha funcionalidad, hemos optado por la misma lógica que al eliminar un producto o una lista de la compra, solo que hemos cambiado la interacción a "DismissDirection.horizontal" el cual permite implementar acciones en las dos direcciones. Por lo tanto, cuando realizamos los endToStart se realiza de manera similar a la anterior, pero si realizamos la acción de StartToEnd, accedemos a la página de creación solo que rellenando los campos con los datos de la lista seleccionada.

```
return Dismissible(
  key: Key(lista.id),
  direction: DismissDirection.horizontal,
  // Permite deslizar en ambas direcciones
  background: Container(
    color: Colors.green,
    alignment: Alignment.centerLeft,
    padding: const EdgeInsets.symmetric(horizontal: 20),
    child: const Icon(Icons.edit, color: Colors.white),
  ),  // Container
  secondaryBackground: Container(
    color: Colors.red,
    alignment: Alignment.centerRight,
    padding: const EdgeInsets.symmetric(horizontal: 20),
    child: const Icon(Icons.delete, color: Colors.white),
  ),  // Container
  confirmDismiss: (direction) async {
    if (direction == DismissDirection.startToEnd) {
      Navigator.push(
        context,
        MaterialPageRoute(
          builder:
              (context) => ListaListaAniadirLista(
                productoOriginal: lista,
                editarLista: (listaEditada) {
                  listaListas.actualizaLista(index, listaEditada);
                  Navigator.pop(context);
                },
                crearLista: (lista) {},
              ),  // ListaListaAniadirLista
        ),  // MaterialPageRoute
      );
      return false;
    }else if (direction == DismissDirection.endToStart) {
      return true;
    }
    return false;
  },
```

(Extra +1,5) Clasificar automáticamente los productos de la lista de la compra según la sección del supermercado en donde se encuentran (panadería, lácteos, frutas, verduras, etc.) Cuando un usuario añada un nuevo producto a la lista, debe aparecer en la categoría adecuada

Para permitir dicha funcionalidad, en la clase ListaCompraPantalla, en la parte superior, se ha implementado un icono, el cual permite indicar de manera intuitiva, un menú desplegable, el cual recibe como parámetros estructuras de tipo List<PopupMenuItem<int>> el cual proporciona una serie de opciones en función a los valores del enumerado, permitiendo seleccionar, y poner en la parte superior, los productos cuya categoría coincidan con la seleccionada en el menú, mediante la función ordenarPorImportancia.

```
PopupMenuButton(
  onSelected: (int item) {
    final manager = Provider.of<ListaCompra>(context, listen: false);
    manager.ordenarPorImportancia(_idLista, item);
  },
  itemBuilder: (BuildContext context) {
    return Importancia.obtenerPopupMenuItemsDesdeEnum();
  },
),  // PopupMenuButton
```

```
enum Importancia {
  General,
  Lacteos,
  Congelados,
  Carnes,
  Frutas,;

  static Importancia getImportanciaDesde({required String nombre}) {
    return Importancia.values.byName(nombre);
  }

  static List<PopupMenuItem<int>> obtenerPopupMenuItemsDesdeEnum() {
    return Importancia.values.map((importancia) {
      return PopupMenuItem<int>(
        value: importancia.index,
        child: Text("Ordenar por: " + importancia.name),
      );  // PopupMenuItem
    }).toList();
  }

}
```

```
void ordenarPorImportancia(String idLista, int valueImportancia) {
  List<int> indices = [];
  for (var i = 0; i < _productos.length; i++) {
    if (_productos[i].listaId == idLista) {
      indices.add(i);
    }
  }

  for (int i = indices.length - 1; i >= 0; i--) {
    if (_productos[indices[i]].importancia.index == valueImportancia &&
        _productos[indices[i]].completado == false) {
      final producto = _productos[indices[i]];

      _productos.removeAt(indices[i]);
      _productos.insert(0,producto.copiaSinNulo());
    }
  }


  _salvarProductos();
  notifyListeners();

}
```