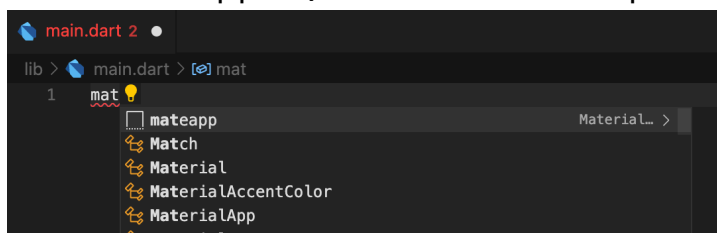


Per a la realització dels exercicis, necessites haver fet totes les passes dels vídeos anteriors. Un cop tinguis tota la infraestructura preparada, prova de generar un projecte i executar-lo a la teva màquina virtual de Android. Si es visualitza correctament i no has tingut cap problema, pot començar amb els exercicis.

A aquesta activitat, programarem una aplicació en Flutter que contengui una gran quantitat de Widgets, ja que ficar-los tots dintre d'una mateixa app, seria difícil. Emprarem els més habituals i que després vos seran útils.

1. Per a començar amb aquest exercici, crearem un nou projecte amb flutter, recordau accedir a les comandes de VSCode > Flutter: New project. Un cop creat el projecte, esborrau tot el contingut de main.dart.

Amb el fitxer buit, emprarem un Snippet (cal tenir instal·lat el pluguin Awesome Snippets), concretament emprarem: *mateapp*



Això ens crearà la interfície necessària per una aplicació bàsica en Flutter.

Finalment canviarem el title tant de MaterialApp com de Scaffold per 'Components'.

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

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Components',
```

```

home: Scaffold(
  appBar: AppBar(
    title: Text('Components'),
  ),
  body: Center(
    child: Container(
      child: Text('Hello World'),
    ),
  ),
),
);
}

```

2. Prova d'executar l'aplicació per si va bé. Recordau eliminar el "banner" de debug dintre de MaterialApp:

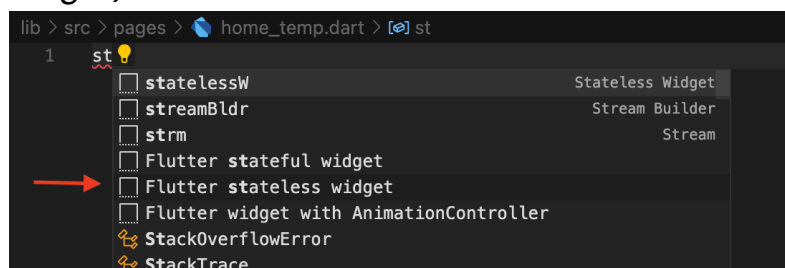
```

debugShowCheckedModeBanner: false,

```

3. Ara anem a crear una nova pàgina. Recordau crear l'estructura de directoris: lib/screens
Dintre del directori pages, cream un nou fitxer anomenat: home_temp.dart
Cream aquest home, en primer lloc per a fer unes proves, després ja implementarem un home, que llegirà d'un fitxer JSON.

En aquest cas, el nostre Home, serà un StatelessWidget. Hi ha un Snippet que ens ajudarà a la creació: *stateless* (Flutter stateless widget).



Anomenam aquest Widget, HomePageTemp. Canviem també el Container que ve per defecte per un Center amb un child de Text. Recordau haver fet l'importació del paquet de Materials.

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

class HomePageTemp extends StatelessWidget {

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Text('Home Temp')
    );
  }
}
```

En el cas que el Snippet que heu emprat vos generi una clau (key), de moment no l'utilitzarem, així que podeu esborrar la línia.

4. Ara ens quedaria modificar el nostre main.dart, per a que enlloc d'emprat un Center(Container(Text... empri la nostra pàgina que hem creat.

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      title: 'Components',
      home: Scaffold(
        appBar: AppBar(
          title: Text('Components'),
        ),
        body: HomePageTemp(
        ),
      ),
    );
  }
}
```

5. Començam amb el primer widget, el ListView! Per això ens dirigim a la pàgina oficial per veure la seva informació:

<https://api.flutter.dev/flutter/widgets/ListView-class.html>

Podem diferenciar entre dos constructors: `ListView` i `ListView.builder`. El primer ens servirà per llistes amb poques quantitats d'informació i estàtiques. El `.builder`, l'utilitzarem quan volguem carregar grans quantitats d'informació i/o aquesta informació és dinàmica, és a dir, va canviant.

Per a programar-lo, ens dirigim dintre de `home_temp.dart`. Esborram el `Center`(`Text`, per al seu lloc, generar un `Scaffold`).

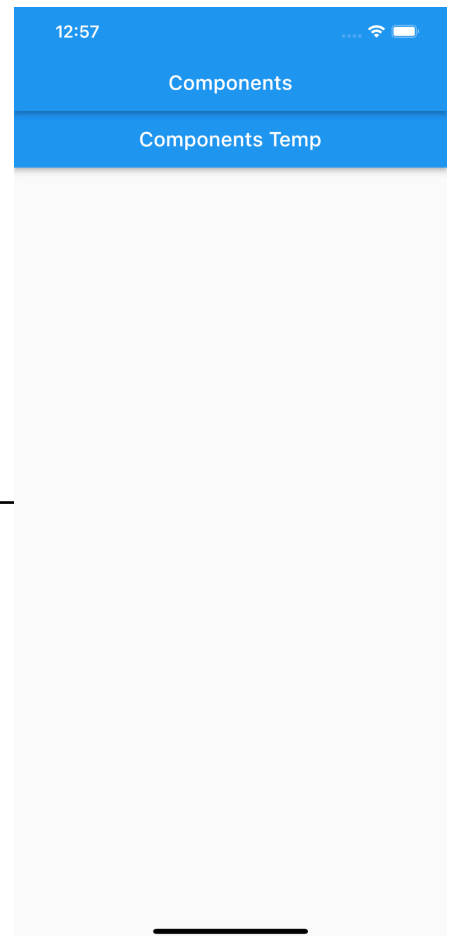
```
class HomePageTemp extends StatelessWidget {  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
    );  
  }  
}
```

Dintre de `Scaffold`, crearem un `AppBar` i també declaram un `body`. modificarem el títol de l'`AppBar`: `Components Temp` i al `body`: hi crearem un `ListView()`.

```
class HomePageTemp extends StatelessWidget {  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(  
        title: Text('Components Temp'),  
      ),  
      body: ListView()  
    );  
  }  
}
```

6. Seguidament, veureu que l'aplicació es mostra amb dues "AppBar". això és degut a que tenim dos widgets de tipus Scaffold anidats. Per a arreglar-ho, eliminarem el Scaffold del main.dart, i sòls hi deixarem el nostre HomePageTemp().
- 7.

```
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      debugShowCheckedModeBanner: false,  
      title: 'Components',  
      home: HomePageTemp()  
    );  
  }  
}
```



8. Anem amb el Widget de ListView. Un ListView és semblant al widget de columnes i files. També té un Array de Widgets. És aquest el que emprarem per anar afegint els nostres widgets.

Normalment dintre d'un ListView s'utilitza el ListTile. Aquest Widget conté altres elements que podem anar modificant. Provem d'afegir-lo, i per cada ListTile li afegim un Text('Títol ListTile'). També podem afegir un Divider() entre els elements de la nostra llista com un Widget independent.

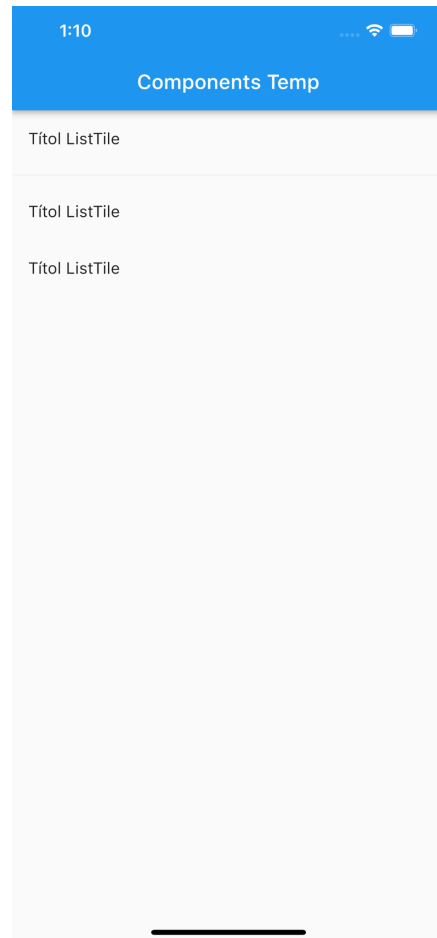
Anem a veure com quedaria:

```

class HomePageTemp extends StatelessWidget {

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Components Temp'),
      ),
      body: ListView(
        children: [
          ListTile(
            title: Text('Títol ListTile'),
          ),
          Divider(),
          ListTile(
            title: Text('Títol ListTile'),
          ),
          ListTile(
            title: Text('Títol ListTile'),
          )
        ],
      ),
    );
  }
}

```



Com podeu observar, hi ha una mica de separació entre el primer element i el segon.

9. Podem també crear la nostra llista a partir d'una llista d'elements. Aquests elements poden ser qualsevol dada que se vos ocorri, de moment per a aquest exemple anem a fer una llista de Strings. Per a fer-ho de forma dinàmica esborrar tot el contingut del children del nostre ListView. Seguidament, crearem una funció que ens retorni una llista de Widgets, i serà el contingut del children.

```

class HomePageTemp extends StatelessWidget {

  final elements = ['Element 1', 'Element 2', 'Element 3', 'Element 4'];

```

```

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text('Components Temp'),
    ),
    body: ListView(
      children: _crearElements(),
    )
  );
}

List<Widget> _crearElements() {
  //Aquesta declaració és la nova degut al null safety
  List<Widget> llista = [];
  return llista;
}

```

Anem ara a assignar el valor per a cada element.

10. Per a crear els elements de la llista ho podem fer de diverses formes, amb un for, o bé amb un forEach. El resultat serà el mateix. Anem a veure les dues formes i també utilitzant els operadors en cascada.

- Utilitzant un for:

```

List<Widget> _crearElements() {
  //Aquesta declaració és la nova degut al null safety
  List<Widget> llista = [];
  for (String element in elements) {
    final tempWidget = ListTile(
      title: Text(element)
    );
    llista.add(tempWidget);
    llista.add(Divider());
  }
  return llista;
}

```

- Amb un for, però emprant operadors en cascada:

```
List<Widget> _crearElements() {  
    //Aquesta declaració és la nova degut al null safety  
    List<Widget> llista = [];  
    for (String element in elements){  
        final tempWidget = ListTile(  
            title: Text(element)  
        );  
        llista..add(tempWidget)  
            ..add(Divider());  
    }  
    return llista;  
}
```

- Utilitzant forEach:

```
List<Widget> _crearElements() {  
    //Aquesta declaració és la nova degut al null safety  
    List<Widget> llista = [];  
    elements.forEach((element) {  
        llista.add(ListTile(  
            title: Text(element),  
        ));  
        llista.add(Divider());  
    });  
    return llista;  
}
```

Com podeu veure, tots tindran el mateix resultat. Afegim un divider entre i entre per a tenir un poc més d'espai.

NOTA: Si voleu fer una prova, posau enlloc de ListTile com a element de la llista directament un Widget de tipus Text. En aquest cas veureu com es visualitza i la necessitat de encapsular les dades dintre de Widgets ja creats.

11. Seguidament, crearem un altre mètode alternatiu que ens permetrà fer el mateix. Es tracta d'utilitzar la funcionalitat Map d'una llista. Aquest realment el que fa és retornar una altra llista a la qual

element per element se li ha aplicat una funció “f”.

```
Iterable<T> map<T>(T toElement(E e)) => MappedIterable<E, T>(this, toElement);
```

```
Iterable<T> map<T>(T Function(String) toElement)
```

dart:core

The current elements of this iterable modified by [toElement].

Returns a new lazy [Iterable] with elements that are created by calling [toElement] on each element of this [Iterable] in iteration order.

The returned iterable is lazy, so it won't iterate the elements of this iterable until it is itself iterated, and then it will apply [toElement] to create one element at a time. The converted elements are not cached. Iterating multiple times over the returned [Iterable] will invoke the supplied [toElement] function once per element for on each iteration.

Methods on the returned iterable are allowed to omit calling [toElement] on any element where the result isn't needed. For example, [elementAt] may call [toElement] only once.

Per fer això declararem el següent:

```
List<Widget> _crearElementsCurt() {  
  // Amb aquest mètode volem emprar la funcionalitat Map de les llistes  
  var widgets = elements.map((element) => null);  
  
  return widgets;  
}
```

Per arribar a aquest codi, realment hem començat a escriure:
elements.ma... i ja ens ha generat la funció.

A aquest codi, entenem que element seria, cada un dels elements de la llista. Ara només ens faltaria veure que fem amb el null. Que creieu que hi va aquí? Normalment després dels símbols “=>” que hi va?

El contingut d'una funció, és a dir les instruccions que s'executaran i podran emprar element com a paràmetre. Realment element sòls és un nom de paràmetre, li podeu posar el nom que vulgueu.

```
List<Widget> _crearElementsCurt() {  
  // Amb aquest mètode volem emprar la funcionalitat Map de les llistes  
  var widgets = elements.map((element) {  
  
  });  
  
  return widgets;  
}
```

Utilitzarem les claus ja que ens resultarà més ordenat per a diverses línies de codi. Aplicam el que realment volem, que és generar un ListTile per a cada element.

```
List<Widget> _crearElementsCurt() {  
  // Amb aquest mètode volem emprar la funcionalitat Map de les llistes  
  var widgets = elements.map((element) {  
    return ListTile(  
      title: Text(element),  
    );  
  });  
  
  return widgets;  
}
```

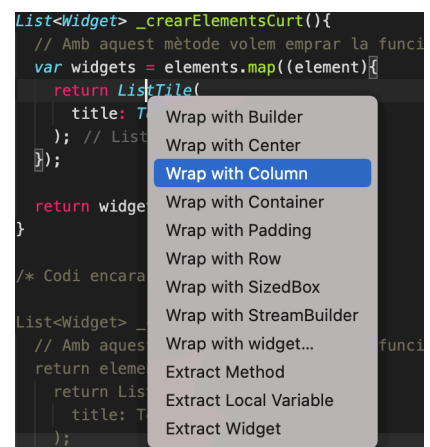
Ara ja només ens quedaria resoldre l'error, i és que volem retornar un element de tipus List<Widget>, però estam generant un Iterable<ListTile>. Per a arreglar-ho, passam widgets a llista.

```
return widgets.toList();
```

Finalment només ens queda modificar el nom del children del ListView.

```
children: _crearElementsCurt(),
```

12. Provau de concatenar un String a element. Per exemple un signe d'exclamació: + "!"
- Seguidament, complicarem una mica l'element de cada llista. Però per a no haver de refer el codi podrem utilitzar una característica de Flutter. Amb la drecera "Ctrl + ." podrem accedir a les opcions de modificació, i utilitzarem la de wrap with column. Això ens permetrà reutilitzar el ListTile, però com a un widget ja existent del Widget Column.



13. D'aquesta forma podrem afegir per cada element de la llista diferents Widgets. Anem a fer-ho! De moment afegirem un Divider per guanyar una mica de separació entre elements. Ara toca explotar una mica més les característiques de ListTile. Si veieu el constructor, conté altres paràmetres a part de title. Proveu d'afegir:

- Un subtítol (Posau-hi text)
- Un leading (Posau-hi una Icona)
- Un trailing (Posau-hi una Icona)

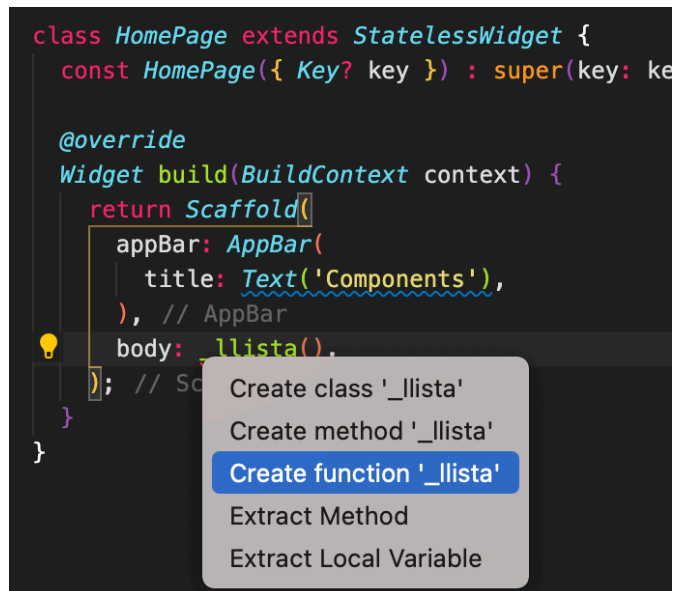
Proveu també d'afegir la característica onTap, veure com podeu fer clicable els elements de la llista.

14. Ara anem a crear el home page definitiu. No és necessari que esborreu el home page que acabam de crear.

Creem un nou fitxer home_page.dart. Fem la importació dels materials.

A continuació creem mitjançant els Snippets, un StatelessWidget anomenat HomePage, que retorni un widget de tipus Scaffold. Implementem el títol ('Components') i al body hi cridem el mètode _llista().

Aquest mètode vos sortirà com a error, ja que no està implementat, per a solucionar-ho, podem declarar-ho, o emprar la dreccera "Ctrl + ." i veurem els missatges que ens apareixen:



```
class HomePage extends StatelessWidget {  
  const HomePage({ Key? key }) : super(key: key);  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(  
        title: Text('Components'),  
      ), // AppBar  
      body: _llista(),  
    ); // Scaffold  
  }  
}
```

The screenshot shows an IDE with a Dart code file. The code defines a `HomePage` class extending `StatelessWidget`. Inside the `build` method, a `Scaffold` widget is returned, containing an `AppBar` with the title 'Components' and a `body` property set to `_llista()`. The `_llista()` method is not yet implemented, which triggers an IDE suggestion menu. The menu offers several options: 'Create class '_llista'', 'Create method '_llista'', 'Create function '_llista'' (which is highlighted in blue), 'Extract Method', and 'Extract Local Variable'.

Seleccionem Create function. Aquesta ens genera la declaració, faltaria incorporar-li el tipus de dada a retornar (Widget).

15. Podreu observar que no apareixen canvis a la pantalla del vostre dispositiu, arribats a aquesta altura ja hauríeu de saber per què. Efectivament, no hem canviat dintre del main, el paràmetre “home” del nostre MaterialApp. Un cop fet, la pantalla apareixerà en blanc i només hi haurà una llista fora elements.

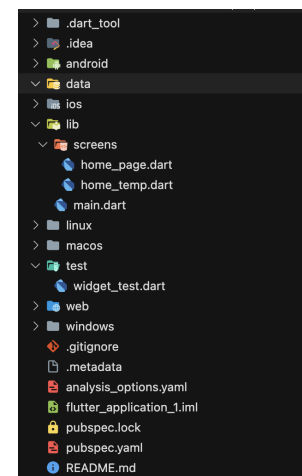
16. Dintre del mètode _llista, especificarem el paràmetre “children” de ListView com una funció que programarem a continuació anomenada _llistaElements(). Aquesta funció haurà de retornar una llista de Widgets. Podeu emprar la drecera anterior per declarar funcions.

```
Widget _llista() {  
  // Retornam un Widget de tipus ListView,  
  // el children del qual és un mètode  
  //que retornarà una llista de Widgets  
  return ListView(  
    children: _llistaElements(),  
  );  
}  
  
List<Widget> _llistaElements() {  
  // Retorna llista buida  
  return [];  
}
```

17. L'objectiu és generar la llista a partir d'un fitxer Json. Per això ara ja tenim l'estructura creada, sòls ens quedarà anar llegint el fitxer i creant els diferents elements. Com podem llegir un fitxer Json?

En primer lloc hem de copiar el fitxer Json al nostre directori del projecte. Per això crearem un directori dintre de l'arrel del projecte, per exemple: “data”.

Un cop carregat, per a que el pugui emprar dintre del projecte, m'he de dirigir al fitxer pubspec.yaml.



Aquest fitxer és molt semblant al Package.json d'un projecte en Node. A aquest fitxer administrarem els paquets, dependències i també els recursos estàtics de la nostra aplicació. En el cas del nostre fitxer Json, és un fitxer estàtic, no canviarà en el temps.

Per a poder-lo utilitzar en el nostre projecte, dintre del fitxer pubspec.yaml identificam la part que posa "assets" Ho trobareu comentat amb #. Per a descomentar-ho, ho podeu esborrar o bé aplicau "Ctrl + / ". A aquest fitxer és molt important les tabulacions.

```
# To add assets to your application, add an assets section, like this:  
assets:  
  - data/menu_opts.json  
#   - images/a_dot_burr.jpeg  
#   - images/a_dot_ham.jpeg
```

Seguidament per

incloure un recurs ho fem mitjançant: guió, espai i la ruta absoluta del fitxer.

Guardau els canvis i veure que es realitza un package get.

Finalment, realitzau un full restart, que implica aturar l'aplicació (recuadre vermell) i tornar a executar (F5).

18. Ara crearem una classe que ens permetrà anar llegint la informació del nostre fitxer Json. Abans crearem un nou directori dintre de lib, anomenat providers. dintre d'aquest directori hi crearem un fitxer anomenat "menu_providers.dart".

Un cop tenim aquesta estructura ens crearem la nostra classe a dintre amb el seu constructor. A més aquesta classe contindrà una llista de tipus dinàmica inicialment buida: List<dynamic> opcions = [];

També al constructor, hi cridarem un mètode anomenat carregarDades().

```
class MenuProvider{  
  List<dynamic> opcions = [];  
  MenuProvider(){  
    CarregarDades();  
  }  
  
  CarregarDades() {
```

```
}  
}
```

Ara per a poder llegir d'un Json, haurem d'importar una llibreria. Concretament utilitzarem: service.dart
Però d'aquesta només emprarem l'objecte rootBundle, això ho podem fer indicant la sintaxi show.

```
import 'package:flutter/services.dart' show rootBundle;
```

Dintre del mètode CarregarDades, utilitzarem el rootBundle per a carregar les dades. Concretament el mètode loadString(ruta del fitxer.json). Aquest mètode realment retorna un future. Recordau que els futures s'executaven en segon pla i utilitzaven una funció amb l'ajuda de ".then". Dintre de then podrem utilitzar el paràmetre de la funció que contindrà la informació llegida del Json. Ens quedaria de la següent forma.

```
CarregarDades() {  
  rootBundle.loadString('data/menu_opts.json').then((data) {  
    print(data);  
  });  
}
```

Finalment, farem la classe privada, i també el seu constructor. A més crearem una instància de la mateixa classe dintre del mateix fitxer i serà aquesta la que emprarem.

```
import 'package:flutter/services.dart' show rootBundle;  
  
class _MenuProvider{  
  List<dynamic> opcions = [];  
  _MenuProvider() {  
    CarregarDades();  
  }  
  
  CarregarDades() {
```

```

rootBundle.loadString('data/menu_opts.json').then((data) {
  print(data);
});
}
}

final menuProvider = new _MenuProvider();

```

Si voleu comprovar si es crea bé la classe i també si es llegeix bé el Json, podem cridar la instància que acabam de crear menuProvider des de HomePage. Per exemple fent un print de la variable opcions. El podeu col·locar com es veu a continuació dintre de LlistaElements.

```

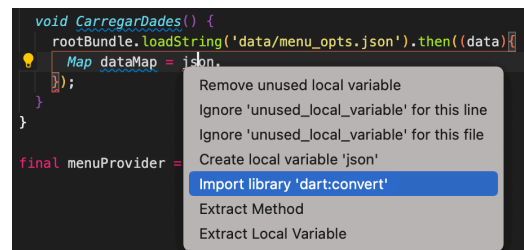
List<Widget> _llistaElements() {
  // Prova menuProvider
  print(menuProvider.opcions);
  // Retorna llista buida
  return [];
}

```

D'aquesta forma veureu pel Debug Console la sortida dels prints que correspon al print del Json que es crida al constructor, i el print de la variable opcions que està buida.

19. Com podeu haver vist, la sortida del Json està en format String, i no és fàcil tractar amb ella. Per a poder tractar la informació del Json correctament, es molt aconsellable utilitzar un Map. Ens centrarem dintre del mètode carregarDades. Creem un nou Map a partir de la variable data.

Per a transformar d'un String a un Map, utilitzarem la funció decode de json. Per això haurem d'importar la llibreria "dart:convert".



```

Map dataMap = json.decode(data);

```

A partir del nostre Map, podem imprimir tot el conjunt, o bé accedir a

diferentes part d'aquest. Per exemple:

```
Map dataMap = json.decode(data);  
print(dataMap);  
print(dataMap['nomApp']);  
print(dataMap['rutes']);
```

PROBLEMS 21 OUTPUT DEBUG CONSOLE TERMINAL

```
Restarted application in 442ms.  
flutter: []  
flutter: {nomApp: Components, rutes: [{ruta: alert, icona: add_alerta, argetes}]}  
flutter: Components  
flutter: [{ruta: alert, icona: add_alert, texte: Alertes}, {ruta: av
```

Com podeu observar les opcions, seran les rutes del nostre fitxer Json. Per això, guardarem dintre d'opcions el dataMap['rutes'].

```
Map dataMap = json.decode(data);  
print(dataMap);  
opcions = dataMap['rutes'];
```

PROBLEMS 19 OUTPUT DEBUG CONSOLE TERMINAL

```
Restarted application in 482ms.  
flutter: []  
flutter: {nomApp: Components, rutes: [{ruta: alert, argetes}]}
```

Però ara tenim un petit problema, i es que la impressió de la variable opcions, ens surt que està buida, però, sí que es llegeix correctament el Json, i també em fa el print de dataMap. Per què?

Aquesta qüestió té molt a veure amb el fet que el mètode loadString és un future. L'execució del print d'opcions es fa abans que la lectura i l'assignació del Json. Com podem solucionar-ho?

```
CarregarDades() async {  
  final resposta = await rootBundle.loadString('data/menu_opts.json');  
  Map dataMap = json.decode(resposta);  
  print(dataMap);  
  opcions = dataMap['rutes'];  
  return opcions;  
}
```

Podem fer que s'esperi a haver llegit el fitxer Json, i a continuació fer l'assignació però igualment no donaria resultat. El més interessant de tot plegat és la funcionalitat que ens brinda el async, que ens

retorna un future. El fet que retorni una informació/tasca que es resoldrà en el futur ens permet treballar amb aplicacions StatelessWidget.

Per això emprarem el Future Builder.

Realment ja no emprarem el carregarDades() de dintre el constructor.

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

class _MenuProvider{
  List<dynamic> opcions = [];
  _MenuProvider(){
    //CarregarDades();
  }

  Future<List<dynamic>> CarregarDades() async {
    final resposta = await rootBundle.loadString('data/menu_opts.json');
    Map dataMap = json.decode(resposta);
    print(dataMap);
    opcions = dataMap['rutes'];
    return opcions;
  }
}

final menuProvider = new _MenuProvider();
```

20. Partint de que disposam d'un mètode CarregarData() que ens retorna un Future, ens dirigim a HomePage, on volem carregar la llista. Per a poder executar un mètode que retorna un future, recordam que havíem de posar l'opció ".then" i després utilitzar el paràmetre de la funció com a les dades que vull utilitzar.

```
Widget _llista() {
  //print(menuProvider.opcions);
  menuProvider.CarregarDades().then((data) {
    print('Llista: ');
    print(data);
  });
}
```

```
return ListView(  
  children: _llistaElements(),  
);  
}
```

Podriem implementar la construcció del list view dintre del future que acabam de cridar?

La resposta és NO, ja que si s'implementa d'aquesta forma, l'aplicació tindria un efecte de "congelat" mentre s'està llegint i carregant la informació del Json, de tal manera que no seria interactiva. Per a evitar aquest fet, Flutter ens ofereix el Future Builder. Anem a esborrar el codi i deixam comentat el mètode:

```
Widget _llista() {  
  // menuProvider.CarregarDades()  
  
  return ListView(  
    children: _llistaElements(),  
  );  
}
```

21. Com funciona un FutureBuilder? És un complement que permet redibuixar-se a si mateix depenent del darrer estat d'un Future.

<https://api.flutter.dev/flutter/widgets/FutureBuilder-class.html>

Anem a veure com s'implementa i un exemple. En primer lloc cal tenir clar que un Future té diferents estats. Anosaltres ens interessa:

- Quan s'està demanant la informació
- Quan s'ha resolt
- Quan dona error

El mètode _llista, ja no retornarà una ListView. Esborram el contingut i declaram la sentència: return FutureBuilder().

Veurem que dona error, que necessita un paràmetre obligatori anomenat builder. A més hi ha altres paràmetres:

- future: va enllaçat al que nosaltres esperam d'aquesta funció
- initialData: seria el valor que prendria a l'inici la nostra llista, a

aquest cas podriem posar una llista nul·la. Aquest paràmetre és opcional.

- builder: és una funció que rep per paràmetre dos arguments, el contexte i un tipus de dada que retorna el future. La sortida es la generació d'un Widget. És aquí on podrem implementar la nostra llista.

```
Widget _llista() {  
  // menuProvider.CarregarDades()  
  return FutureBuilder(  
    future: menuProvider.CarregarDades(),  
    initialData: [],  
    builder: (context, AsyncSnapshot<dynamic> snapshot)  
  ),  
};  
  
// return ListView(  
//   children: _llistaElements(),  
// );
```

Podem declarar el tipus snapshot com a

AsyncSnapshot, però

convindria especificar el tipus de dada que emprarem.

Per això, especifícam <List<dynamic>> després de AsyncSnapshot. Només queda retornar el widget de tipus ListView que ja teniem definit abans.

```
Widget _llista() {  
  // menuProvider.CarregarDades()  
  return FutureBuilder(  
    future: menuProvider.CarregarDades(),  
    initialData: [], // Aquest seria el valor per defecte que s'envia a  
    snapshot.data  
    builder: (context, AsyncSnapshot<List<dynamic>> snapshot){  
      print('builder');  
      print(snapshot.data);  
      return ListView(  
        children: _llistaElements(),  
      );  
    },  
  );  
}
```

Veieu alguna cosa estranya en els prints?

Afegiu, com a paràmetre de _llistaElements: snapshot.data

A continuació anem a programar _llistaElements(List<dynamic> data).

En primer lloc declaram una variable amb final de tipus `List<Widget>` anomenada `elements`, i la inicialitzam a llista buida `[]`.

Seguidament, amb el paràmetre `data`, realitzam un `forEach`, de tal forma que anirem llegint cada element de `data`. Per cada element de `data`, anirem creant un `ListTile` i un `Divider` i els afegirem a la llista d'element.

Al `ListTile` li afegirem els paràmetres, `title`, `leading`, `trailing` i `onTap`. El `title` hauria de ser el text que apareix per a cada element del nostre fitxer `Json`:

```
"nomApp" : "Components",
"rutes" : [
  {
    "ruta" : "alert",
    "icona" : "add_alert",
    "texte": "Alertes"
  }
]
```

```
title: Text(element['texte']),
```

Finalment retornarem aquesta llista.

```
List<Widget> _llistaElements( List<dynamic>? data ) {
  final List<Widget> elements = [];
  data?.forEach((element) {
    final widgetTemp = ListTile(
      title: Text(element['texte']),
      leading: Icon(Icons.account_circle, color: Colors.blue,),
      trailing: Icon(Icons.keyboard_arrow_right, color: Colors.blue),
      onTap: () {},
    );
    elements..add(widgetTemp)
              ..add(Divider());
  });
  return elements;
}
```

22. Ara quedaria carregar les icones a partir del descriptor que apareix al `Json`, `"icona"`. Per a fer això, no és possible llegir un `String` i adjuntar-lo a `Icons` per a cercar-lo, o alguna mena de mètode `toString` per a `Icons`. Hi ha alguna llibreria que ens ajudaria a resoldre-ho, com per exemple `icons_helper`.

https://pub.dev/packages/icons_helper/example

Aquesta està desenvolupada per `voilaireapp.com`, però encara està un poc verda.

Per això, nosaltres crearem el nostre propi mecanisme per a fer-ho i

així aprendrem més conceptes.

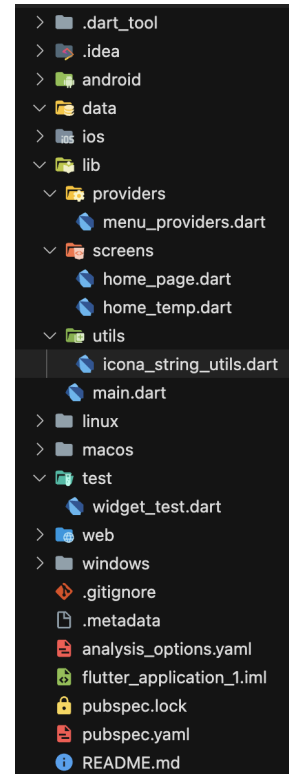
23. En primer lloc crearem un nou directori dintre de lib, anomenat "utils". Aquí com el nom indica hi anirem col·locant les classes que emprem com a utilitats, eines i que emprarem de forma global en el projecte.

Dintre de la carpeta hi crearem un fitxer anomenat: `icono_string_utils.dart`

Començam a programar aquesta classe.

Declararem un mètode anomenat "getIcon" que retorni una variable de tipus *Icon* a partir d'un String.

De moment deixam amb un `return Icon()` que dona error, però després el completarem.



```
Icon getIcon(String nomIcona) {  
  return Icon();  
}
```

També declaram un Map de icones i el declaram en mode privat.

```
final _icons = <String, IconData>{  
  
};
```

Aquest mapa realment tindrà una tupla String i IconData, on a String hi tindreu la clau valor que hi ha al nostre JSON i com a IconData la definició de Icons.(el valor de la icona). Anem a veure com quedaria per a les icones del nostre JSON.

```
final _icons = <String, IconData>{  
  'add_alert'      : Icons.add_alert,  
  'accessibility'  : Icons.accessibility,
```

```
'folder_open' : Icons.folder_open  
};
```

D'aquesta forma ja tenim definits els valor per cada clau. Ara només ens quedaria modificar el mètode getIcon

```
Icon getIcon(String nomIcona) {  
  return Icon(_icons[nomIcona], color: Colors.blue);  
}
```

Com podem veure, realment aquesta seria la definició d'una Icona, talment com la podríem tenir a home_page.dart. Però es generarà a partir d'un String. Ara ja només ens quedaria cridar aquest mètode des del *leading*: del ListTile del home_page. Pensau a realitzar la importació de la classe.

```
leading: getIcon(element['icona']),
```

Si en aquesta part vos dona error revisau que el fitxer JSON, a la part de del descriptor de la icona, té aquesta paraula i no una altre. Finalment comprovau que les icones canvien.

```
"nomApp" : "Components",  
"rutes" : [  
  {  
    "ruta" : "alert",  
    "icona" : "add_alert",  
    "text": "Alertes"  
  },  
  ...  
]
```

NOTA: Realment aquesta funcionalitat que acabam de crear té una pega, la sabrieu dir?

Efectivament, realment només ens generarà les icones que estiguin mapejades a la nostra variable. Per això si voleu intentar emprar la utilitat abans esmentada, o bé voleu esbrinar un poc sobre noves funcionalitats, sentiu-vos lliures en aquesta part de fer-ho.

24. Ara ja per acabar aquesta part, ens quedaria fer el següent.

Permetre a Flutter, obrir una nova pàgina quan realitzem una acció, com per exemple pulsar un botó, una opció, etc i permetre navegar entre les diferents pàgines. En l'Android Studio es coneixen com Intents.

Aquesta acció la implementarem dintre del mètode onTap: del

listTile de moment. En primer lloc anem a crear noves pàgines dintre del directori pages. De moment una per Alert i l'altre per Avatar. De moment el Card ho deixau.

Dintre del directori pages, cream un fitxer alert_page.dart i un altre anomenat avatar_page.dart.

Començam pel alert_page.dart:

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

class AlertPage extends StatelessWidget {

  @override
  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        title: Text('Pàgina Alert'),

      ),

    );

  }

}
```

Declaram un StatelessWidget anomenat AlertPage, implementam el build i retornam un Scaffold. Ara ja sabeu com anar-ho implementant, acabam de definir una appBar, que tingui un títol descriptiu. Podeu fer un copiar i aferrar a l'altre fitxer, canviant el nom de la classe i el títol de l'appBar.

Com podem fer per accedir a aquestes pàgines? Per a fer-ho dirigiu-vos a home_page, i dintre de onTap() {}, declararem.

Per a fer un “push” de la nova pàgina necessit l'objecte “Navigator”. Aquest objecte, està inclòs en el paquet material i un del seus mètodes és el *push()*.

```
onTap: () {
```

```
Widget _llista() {
  // menuProvider.CarregarDades()
  return FutureBuilder(
    future: menuProvider.CarregarDades(),
    initialData: [],
    builder: (context, AsyncSnapshot<List<dynamic>> snapshot){
      // print('builder');
      // print(snapshot.data);
      return ListView(
        children: _llistaElements(context, snapshot.data ),
      ); // ListView
    },
  ); // FutureBuilder
}

List<Widget> _llistaElements(BuildContext context, List<dynamic>? data ) {
  final List<Widget> elements = [];
  data?.forEach((element) {
    final widgetTemp = ListTile(
      title: Text(element['texte']),
      leading: getIcon(element['icona']),
      trailing: Icon(Icons.keyboard_arrow_right, color: Colors.blue),
      onTap: (){
        Navigator.push(context, route)
      },
    ); // ListTile
  });
}
```

```
Navigator.push(context, route)
},
```

Com podem veure té dos paràmetres. El context, és necessari per saber l'estat de l'aplicació, conté informació global d'aquesta, i ens serà d'utilitat quan volem navegar d'una pàgina a una altre per saber quina és la pàgina "activa". Realment aquest context ja el tenim definit, però l'hem de fer arribar a la nostra funció `_llistaElements()` d'alguna forma.

Si vos fixau, el future builder ja té un context, per tant el podriem afegir de d'allà al mètode `_llistaElements()` i ja no tenim l'error al context.

I ara per a la ruta? En primer lloc crearem una variable final anomenada ruta per a fer-ho un poc més senzill ja que sinó quedarà massa codi dintre de `push()`.

Realment és millor emprar rutes per nom, que són reutilitzables, però en primer lloc veurem les rutes estàtiques.

Aquesta variable, tindrà la definició d'un objecte anomenat `MaterialPageRoute()`, on el seu builder retornarà la pàgina desitjada a la qual volem anar. Així ja tindrem resolt

```
onTap: () {
  final route = MaterialPageRoute(builder: (context) {
    return AlertPage();
  });
  Navigator.push(context, route);
},
```

Una altre forma de definir-ho, amb sintaxi simplificada seria:

```
onTap: () {
  /*
  final route = MaterialPageRoute(builder: (context) {
    return AlertPage();
  });*/
```



```
final route = MaterialPageRoute(builder: (context) => AlertPage());
Navigator.push(context, route);
},
```

Si vos fixau i ho provau, tots els botons, ara ens portaran a la mateixa pàgina. Això ho podríem solucionar fent un condicional aquí dintre però no és la solució més òptima i en el següent punt veurem com fer-ho.

Cal dir que també podríem enviar dades a les noves pàgines com a paràmetres nombrats i/o posicionals, per enviar informació a aquestes, etc.

Ara bé, abans de passar al següent punt, anem a veure com podem tornar enrere, ja sabem que podem utilitzar el botó del sistema, però i si vull un botó aposta per a tornar a la pàgina anterior dintre de la meua pàgina?

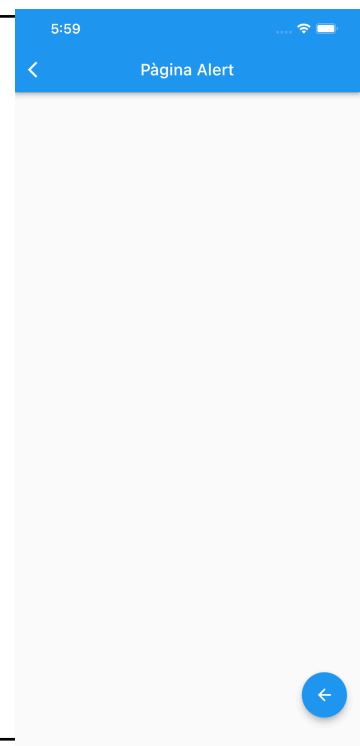
Anem a definir un botó dintre de la pàgina. Per a fer això declaram un `floatingActionButton` en el nostre Scaffold. El mètode `onPressed()` {} del botó, només haurà d'executar la funció

`Navigator.pop(context);`

Podeu posar la icona que vulgueu.

```
class AlertPage extends StatelessWidget {

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Pàgina Alert'),
      ),
      floatingActionButton: FloatingActionButton(
        child: Icon(Icons.arrow_back),
        onPressed: () {
          Navigator.pop(context);
        }
      ),
    );
  }
}
```



25. Quan tenim poques pàgines, la forma que hem vist anteriorment, és vàlida, però quan volem establir moltes rutes i a més de forma dinàmica, necessitam una forma diferent. Navegar amb rutes nombrades. Com les podem implementar?

Farem feina dintre del `onTap()` del `ListTile`, podeu comentar el codi anterior, però no l'esborreu.

El mètode `Navigator.pushNamed` té dos paràmetres, el context i un `String` que és la pàgina a la qual navegar. Per exemple en web seria alguna cosa com:

```
Navigator.pushNamed(context, '/home');
```

Si aquí posam la ruta que prové del nostre fitxer JSON, vos funciona?

```
Navigator.pushNamed(context, element['ruta']);
```

Efectivament, no funciona, falta alguna cosa més a fer i es que la nostra app no sap on dirigir-se quan rep una ruta determinada. ON podem definir aquestes rutes? De moment ho farem dintre del nostre main.

Comentam el home: `HomePage()` i declaram el paràmetre `routes`:

```
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      debugShowCheckedModeBanner: false,  
      title: 'Components',  
      // home: HomePage(),  
      routes: ,  
    );  
  }  
}
```

Si posau el cursor a sobre del `routes` veureu que vos demanda un atribut de tipus `Map`, compostat per una clau valor, `String` i una funció

que rep un BuildContext i retorna un Widget.

```
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      debugShowCheckedModeBanner: false,  
      title: 'Components',  
      // home: HomePage(),  
      initialRoute: '/',  
      routes: <String, WidgetBuilder>{  
        '/'      : (BuildContext context) => HomePage(),  
      },  
    );  
  }  
}
```

Com podeu veure a continuació, tenim la clau definida per un String, en aquest cas la barra d'inici representativa de la navegació web. Si realitzau un host reload, veure que ja vos torna anar bé l'aplicació però per què?

Per a fer-ho d'una forma correcta, hauriem de definir la ruta inicial.

```
initialRoute: '/',  
routes: <String, WidgetBuilder>{  
  '/'      : (BuildContext context) => HomePage(),  
},
```

Ara de moment encara no puc anar a cap ruta amb els botons. Per a fer-ho afegirem dues rutes més al Map de routes.

```
routes: <String, WidgetBuilder>{  
  '/'      : (BuildContext context) => HomePage(),  
  'alert'   : (BuildContext context) => AlertPage(),  
  'avatar'  : (BuildContext context) => AvatarPage(),  
},
```

Més envant ja separarem les rutes del main, però de moment ho feim aquí.

Fixau-vos que el que realment estam fent, és per cada botó executar

un “pushNamed” amb la referència de la ruta que llegim del JSON. Aquesta ha de coincidir al nostre Map de routes, sinó la ruta donaria error. Recordau que encara no hem implementat la ruta de targetes

La majoria d'aplicacions té més d'una pantalla i aquest mecanisme que hem vist és quasi necessari en qualsevol aplicació. Recordar que també a través d'aquest podrem enviar dades o informació a les altres pàgines mitjançant el constructor.

.

A continuació veurem com tractar excepcions.

26. Quan polsam el botó de Card- Targetes, estam dirigint-nos a la pàgina `"ruta" : "card"` i aquesta no existeix, no està definida a l'apartat de rutes. Doncs, què passa? Que podriem fer per a poder definir per exemple una acció per defecte quan rebem una ruta que no existeix?

Si mirau les característiques del MaterialApp té un paràmetre anomenat “onGenerateRoute”: `{Route<dynamic>? Function(RouteSettings)? onGenerateRoute}`

Si vos fixau, com a valor del paràmetre hem de declarar una funció que rebrà per paràmetre una variable de tipus RouteSettings i haurà de retornar una variable de tipus Route<dynamic>.

Anem a programar-ho.

```
onGenerateRoute: (RouteSettings settings) {  
  print('Hem anat a: ${settings.name}');  
},
```

D'aquesta forma, encara no ens dirigirà enlloc quan polsam “Cards-targetes” però si que ens haurà d'imprimir el nom d'on es vol anar.

Veieu el print? Ara anem a retornar una ruta vàlida. Per a fer-ho utilitzarem la classe MaterialPageRoute, el constructor de la qual ens redirigirà a una pàgina ja creada, per exemple AlertPage().

```
onGenerateRoute: (RouteSettings settings) {
```

```

    print('Hem anat a: ${settings.name}');
    return MaterialPageRoute(
      builder: (BuildContext context) => AlertPage()
    );
  },

```

D'aquesta forma, tidriem com recollir les excepcions, o bé com una pàgina per defecte.

NOTA: Si tenim una gran quantitat de rutes, les hauriem de separar del main. A continuació veurem com fer-ho.

27. Crearem un nou directori anomenat "routes" dintre de "lib". Dintre d'aquest directori cream, un fitxer anomenat "routes.dart". A continuació copiam i esborram el Map de String i WidgetBuilder que està com a paràmetre routes i l'aferram dintre d'aquest nou fitxer. Declarau-ho com una variable final routes. Pensau a fer els imports necessaris.

```

import 'package:exercici2_mix/screens/alert_page.dart';
import 'package:exercici2_mix/screens/avatar_page.dart';
import 'package:exercici2_mix/screens/home_page.dart';
import 'package:flutter/material.dart';

final routes = <String, WidgetBuilder>{
  '/'      : (BuildContext context) => HomePage(),
  'alert'   : (BuildContext context) => AlertPage(),
  'avatar'  : (BuildContext context) => AvatarPage(),
};

```

Realment aquesta forma de declarar les rutes no és del tot pràctica, podem definir aquí una funció que retorni el mateix tipus de dada i retorni aquest valor. D'aquesta forma, des de la classe main només haurem de cridar al mètode.

```

Map<String, WidgetBuilder> getRoutes() {
  return <String, WidgetBuilder>{
    '/'      : (BuildContext context) => HomePage(),
    'alert'   : (BuildContext context) => AlertPage(),
  };
}

```

```
'avatar'      : (BuildContext context) => AvatarPage(),  
  );  
}
```

Ara finalment ens queda cridar al mètode des del main. Comprovau el funcionament.

```
routes: getRoutes(),
```

Un cop enllestida aquesta part, anem a començar a definir cada una de les pàgines.

Seguirem la segona part just en acabar aquesta primera part.