Flutter

Version : 1.0.0

Introduction

Dans ce cours vous allez apprendre les bases du Framework Flutter ainsi que les widgets.

Objectifs pédagogiques

- Apprendre les bases de Flutter
- Concevoir des interfaces utilisateur avec Flutter en utilisant des widgets.

Prérequis

- Base avec le langage Dart

Outils nécessaires

- Flutter SDK
- Un émulateur ou appareil Android

Création d’un projet Flutter

```
flutter create myapp
cd myapp
flutter run
```

/!\ vous devez avoir choisi votre mobile virtuel ou d’avoir branché votre téléphone Android.

Architecture des dossiers

Un projet classique Flutter va contenir 4 dossiers :

- android & iOS : il s'agit des fichiers compilés de votre projet.
- lib : ce dossier va contenir l'ensemble de tous vos fichiers de code Dart
- test : ce dossier va contenir les fichiers de test unitaire de vos code Dart

Widgets

Avec Flutter, chaque partie de l'interface utilisateur est un widget.
Par exemple, sur ma page de démo de Flutter, nous avons 4 widgets :

- Le bandeau de titre
- Le texte au centre
- Le conteur
- Le bouton incrémental

Nous commencerons toujours une page Flutter avec les lignes suivantes :

```dart
// On importe Flutter dans notre code Dart
import 'package:flutter/material.dart';

// On démarre notre application au lancement
void main() => runApp(MyApp());
```

MyApp et MaterialApp

La classe ***MyApp*** va contenir la méthode ***build***() qui permettra de dessiner l'interface utilisateur avec le widget ***MaterialApp***()

Nous donnons ensuite un titre à notre application avec le champ ***title***, et nous mettons également l'attribut ***debugShowCheckedModeBanner*** à false pour supprimer la bannière de débogage.

Enfin nous avons le champ home qui va nous permettre de définir le Widget qui sera la page d'accueil de l'application.

Nous avons également le champ ***theme*** qui utilise ***ThemeData*** pour créer un thème qui va s'appliquer à toute notre application.
Pour ne pas à avoir à rentrer les codes couleurs, nous utilisons la classe ***Colors*** pour récupérer une couleur.

```dart
class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.pink,
        visualDensity: VisualDensity.adaptivePlatformDensity,
      ), // ThemeData
      home: const MyHomePage(title: 'Flutter Demo Home Page'),
      debugShowCheckedModeBanner: false,
    ); // MaterialApp
  }
}
```

Scaffold

La méthode ***build*** de notre ***HomePage*** retourne un ***Scaffold***() qui est le widget qui implémente la structure de base d'une page en utilisant Material Design.

Il nous permettra de définir notre appBar, notre body et ainsi qu'une floatingActionButton.

```
class _MyHomePageState extends State<MyHomePage> {
    @override
    Widget build(BuildContext context) {
        return Scaffold(
            appBar: AppBar(
                title: Text(widget.title),
            ),
            body: Center(
                child: Column(
                    mainAxisAlignment: MainAxisAlignment.center,
                    children: <Widget>[
                        Text(
                            'You have pushed the button this many times:',
                        ),
                        Text(
                            '$_counter',
                            style: Theme.of(context).textTheme.headline4,
                        ),
                    ],
                ),
            ),
            floatingActionButton: FloatingActionButton(
                onPressed: _incrementCounter,
                tooltip: 'Increment',
                child: Icon(Icons.add),
            ),
        );
    }
}
```

Stateful & Stateless

Il existe deux catégories de widget :

- Sans état (Stateless)
  - Nous avons une seule classe qui étends la classe StatelessWidget.
  - Le widget est créé avec le build. Il ne changera jamais d'apparence dans l'UI.

```
class MyHomePage extends StatelessWidget {
    @override
    Widget build(BuildContext context) {
        return Scaffold(
            appBar: AppBar(
                title: Text('Home'),
            ),
        );
    }
}
```

Stateful & Stateless

- Avec état (Stateful)
    - Nous avons deux classes :
        - La première classe étends la classe StatefulWidget. Elle appellera la fonction createState qui permettra de crée notre deuxième classe.
        - La seconde classe étends la classe State<> qui prendra la première classe en tant que type. Cette classe aura alors accès à certaine méthode pour manipuler les états.
    - Le widget est désormais créé avec le build mais est également rebuild à chaque changement du State. Son apparence dans l'UI peut être changé.

Stateful

```dart
class MyHomePage extends StatefulWidget {
    @override
    _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
    @override
    Widget build(BuildContext context) {
        return Scaffold(
            appBar: AppBar(
                title: Text('Home'),
            ),
        );
    }
}
```

SetState

Lorsque l'on souhaite modifier une variable qui est affiché dans l'UI, on utilise la fonction setState pour que le widget soit reconstruit.

```
int _counter = 0;

void _incrementCounter() {
    setState((){
        _counter++;
    });
}
```

Construction de la page de connexion

Supprimer tout le contenu du fichier ***main.dart*** et recommencer du début.

On va commencer par recréer un widget racine, ce qui nous donne :

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

Run | Debug | Profile
void main() => runApp(MyApp());

// Raccourci : Saisisser 'stless' puis appuyer sur Enter
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```

Ajout du MaterialApp

La deuxième étape de va être de modifier le ***Container*** par notre application de rendu ***MaterialApp***.

Nous utiliserons également le widget ***ThemeData*** pour modifier le thème de tous nos widgets, vous pouvez aller voir la doc. pour plus de personnalisation.

Cela nous rend quelque chose comme cela :

```dart
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      title: 'Time Tracker',
      theme: ThemeData(
        primaryColor: Colors.indigo,
      ), // ThemeData
      home: Container(
        color: Colors.white,
      ), // Container
    ); // MaterialApp
  }
}
```

Créer notre première page

Pour avoir un code plus propre et ordonné, chaque widget aura son propre fichier. Ces fichiers seront placer dans leur dossier nommée ***app.classname*** à l'intérieur du dossier lib.

Nous allons désormais créer un nouveau dossier ***app.sign_in*** ainsi qu'un fichier ***sign_in_page.dart*** à l'intérieur de celui-ci.

Nous allons ensuite crée comme précédemment une simple page sans état avec un widget ***Scaffold***.

Nous aurons alors le code suivant :

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

class SignInPage extends StatelessWidget {
  const SignInPage({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Time Tracker'),
        centerTitle: true,
        elevation: 2.0,
      ), // AppBar
    ); // Scaffold
  }
}
```

Importons notre nouveau widget dans le fichier main.

```dart
import 'package:flutter/material.dart';
import 'package:myapp/app.sign_in/sign_in_page.dart';

Run | Debug | Profile
void main() => runApp(MyApp());

// Raccourci : Saisisser 'stless' puis appuyer sur Enter
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      title: 'Time Tracker',
      theme: ThemeData(
        primaryColor: Colors.indigo,
      ), // ThemeData
      home:  const SignInPage(),
    ); // MaterialApp
  }
}
```

Construction de la page de connexion

Avant de coder quoi que ce soit, il faut se demander comment les différents widgets sont construits.

Il faut premièrement les aligner en colonnes et régler l'espace entre chaque bloc, il faudra ensuite leurs attribuer un titre, une couleur et pour finir donner un évènement.

Maintenant que l'on connaît les différentes étapes, on va pouvoir crée les différents widgets.

Le widget Column

Nous allons commencer par ajouter un Container au body qui contiendra tous nos widgets.
On lui donne une couleur pour voir les changements.

```
return Scaffold(
  appBar: AppBar(
    title: Text('Time Tracker'),
    centerTitle: true,
    elevation: 2.0,
  ), // AppBar
  body: Container(
    color: Colors.yellow,
  ), // Container
); // Scaffold
```

Le widget Column

On va maintenant pouvoir ajouter un widget ***Column*** comme enfant de notre containeur qui contiendra nos éléments sous la forme d'une colonne.
Nous utiliserons le paramètres children de ***Column*** pour donner la liste de nos widgets à notre colonne.

```
body: Container(
  color: Colors.yellow,
  child:  Column(children: <Widget>[]),
), // Container
```

Comme exemple, nous pouvons utiliser deux containeurs de couleur différente.
Cependant, un containeur à une taille par défault de 0.
Nous allons donc leurs passer en enfant une SizedBox.
Ce widget possède deux paramètres : width et height, pour pouvoir régler la taille de la boîte.

```
children: <Widget>[
  Container(
    color: Colors.orange,
    child: const SizedBox(width: 100.0, height: 100.0),
  ), // Container
  Container(
    color: Colors.red,
    child: const SizedBox(width: 100.0, height: 100.0),
  ), // Container
], // <Widget>[]
```

Alignement

Nous avons désormais nos widgets sous la forme d'une colonne cependant celle-ci est collé en haut à gauche.
Nous allons alors utilisé les paramètres ***crossAxisAlignment*** pour l'alignement horizontal et ***mainAxisAlignment*** pour l'alignement vertical.
Chacun des paramètres prendra une valeur dans l'énumérateur de même nom.

```
  child: Column(
    mainAxisAlignment : MainAxisAlignment.center,
    crossAxisAlignment : CrossAxisAlignment.stretch,
    children : <Widget>[
        //...
    ],
  ),
),
```

Extraction dans une méthode

Lorsque nous codons, nous somme amener à créer un widget directement dans notre classe, cependant pour plus de lisibilité, il est préférable de créer une autre classe qu'on appellera.

Pour rapidement exporter une classe, il suffit de cliquer sur le widget à exporter puis d'utiliser le raccourci : Ctrl+Shift+R puis de sélectionner Export to Widget dans le menu déroulant.

Marge entre et autour des blocs

Nous allons maintenant ajouter des marges autour de nos blocs avec les paramètres padding, celuici prendra un objet EdgeInsets en arguments.
Comme nous voulons ajouter une marge uniforme autour de nos blocs, nous utiliserons la méthode all de la classe.

Pour plus de sens, nous allons remplacer le widget Container par un widget Padding, il faudra cependant retirer le paramètre color qui n'existe plus, nous n'en auront d'ailleurs plus besoin.

```
body: Padding(
  padding: const EdgeInsets.all(16.0),
  child: Column(
      mainAxisAlignment: MainAxisAlignment.center,
      crossAxisAlignment: CrossAxisAlignment.stretch,
      children: <Widget>[
        Container(
          color: Colors.orange,
          child: const SizedBox(width: 100.0, height: 100.0),
        ), // Container
        Container(
          color: Colors.red,
          child: const SizedBox(width: 100.0, height: 100.0),
        ) // Container
      ]), // <Widget>[] // Column
), // Padding
```

Pour les marges entre les blocs, il suffit simplement d'ajouter des SizedBox avec une hauteur fixe entre nos blocs.

```
Container(
  color: Colors.orange,
  child: const SizedBox(width: 100.0, height: 100.0),
), // Container
const SizedBox(height: 8.0),
Container(
  color: Colors.red,
  child: const SizedBox(width: 100.0, height: 100.0),
), // Container
const SizedBox(height: 8.0),
Container(
  color: Colors.purple,
  child: const SizedBox(width: 100.0, height: 100.0),
), // Container
```

Text, TextStyle et FontWeight

Maintenant que nous allons pouvoir crée un widget Text qui contiendra le texte 'Sign in'.

Pour centrer le texte, nous allons utiliser le paramètre textAlign qui prend comme un toujours un énumérateur de même nom.

Pour finir, nous allons également donner un style à notre texte avec le paramètre style qui prend en aargument un widget TextStyle.

Avec ce dernier, nous allons pouvoir régler la taille du texte ainsi que son épaisseur avec les paramètres fontSize et fontWeight.

```
title: const Text(
  'Sign in',
  textAlign: TextAlign.center,
  style: TextStyle(
    fontSize: 32.0,
    fontWeight: FontWeight.w600,
  ), // TextStyle
), // Text
```

Boutons

Maintenant que nous avons vu comment fonctionnait les widgets, nous pouvons supprimer tous nos conteneurs colorés.

Nous allons désormais crée un bouton , nous utiliserons alors le widget ***ElevatedButton*** qui contient déjà un peu de style.

Notre ajouterons ensuite un texte à notre bouton en tant qu'enfant ainsi qu'un évènement lorsqu'il est pressé avec le paramètre ***onPressed***.
onPressed prend une fonction de type void qui ne prend aucun argument.
La syntaxe classeur pour ce type de fonction est () {}.

Nous pouvons voir apparaître notre message dans la console à chaque qu'on appui sur le bouton.

```
ElevatedButton(
  child: const Text('Sign in with Google'),
  onPressed: () {
    print('button pressed');
  },
), // ElevatedButton
```

Couleurs

Nous avons désormais un bouton, nous allons pouvoir modifier sa couleur avec le paramètre style qui prendra la méthode ***ElevatedButton.styleFrom***.
Nous pourrons modifier la couleur de fond avec le paramètre primary et la couleur d'animation avec onPrimary.

La classe Colors permet aussi d'avoir des dégradés de couleur en rentrant entre crochet un nombre après la couleur.

```
style: ElevatedButton.styleFrom(
  foregroundColor: Colors.grey[400],
  backgroundColor: Colors.white,
),
```

Bordures

Nous pouvons modifier les bordures des boutons avec les paramètres de style ***shape***.
Pour avoir des bordures arrondis sur tous les angles nous utilisons les classes comme ci-dessous. Dans l'exemple nous appliquer un arrondi de 4 sur tous les angles sur notre bouton.

```
shape: const RoundedRectangleBorder(
  borderRadius: BorderRadius.all(
    Radius.circular(4.0),
  ), // BorderRadius.all
), // RoundedRectangleBorder
```

Boutons réutilisables

Nous allons devoir crée boutons pour notre page, le code sera cependant très similaire. nous allons donc crée un modèle de notre bouton dans lequel on ajoutera simplement nos variables.

Pour cela on va créer un nouveau dossier dans lib que l'on nommera common_widgets. Nous créons ensuite un fichier nommé custom_elevated_button.dart.

Nous pouvons ensuite copier notre bouton dans notre fichier, en créant notre classe sans état au passage.

Nous allons ensuite crée les variables et le constructeur sur ce qui doit pouvoir être adapter. C'est à dire : le texte, la couleur, le rayon de la bordure et l'évènement du bouton.

Ce qui nous donne donc :

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

class CustomElevatedButton extends StatelessWidget {
  const CustomElevatedButton({
    super.key,
    required this.child,
    required this.color,
    this.borderRadius = 2.0,
    required this.onPressed,
  });

  final Widget child;
  final Color color;
  final double borderRadius;
  final VoidCallback onPressed;

  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      child: child,
      style: ElevatedButton.styleFrom(
        foregroundColor: Colors.grey[400],
        backgroundColor: color,
        shape: RoundedRectangleBorder(
          borderRadius: BorderRadius.all(Radius.circular(borderRadius)),
        ), // RoundedRectangleBorder
      ),
      onPressed: onPressed,
    ); // ElevatedButton
  }
}
```

De l'autre côté, après avoir importé notre widget, cela donne :

```
CustomElevatedButton(
  color: Colors.white,
  borderRadius: 4.0,
  onPressed: () {},
  child: const Text(
    'Sign in with Google',
    style: TextStyle(
      color: Colors.black87,
      fontSize: 15.0,
    ), // TextStyle
  ), // Text
), // CustomElevatedButton
```

/!\ on n'oublie pas l'import de ***CustomElevatedButton***

Nous allons maintenant pouvoir recommencer une nouvelle fois en créant le widget SignInButton qui sera cette fois-ci spécifique à notre page, nous le placerons donc dans un fichier sign_in_button.dart dans le dossier app.sign_in.

```dart
import 'package:flutter/material.dart';
import 'package:myapp/common_widgets/custom_elevated_button.dart';

class SignInButton extends CustomElevatedButton {
  SignInButton({
    super.key,
    required String text,
    required Color color,
    required Color textColor,
    required VoidCallback onPressed,
  }) : super(
          child: Text(
            text,
            style: TextStyle(
              color: textColor,
              fontSize: 15.0,
            ), // TextStyle
          ), // Text
          color: color,
          onPressed: onPressed,
        );
}
```

Cette fois-ci nous allons simplifier le texte à sa chaîne et a sa couleur. Nous donnerons également une valeur par défaut à notre couleur secondaire.

```
child: SignInButton(
  text: 'Sign in with Google',
  textColor: Colors.black87,
  color: Colors.white,
  onPressed: () {},
), // SignInButton
```

Taille

Nous allons rajouter un paramètre à notre ***CustomElevatedButton*** pour pouvoir régler la taille. Cependant, pour régler la taille nous besoin d'une SizedBox. Nous allons donc déplacer notre ElevatedButton dans une SizedBox.

Pour cela cliquer simplement sur le widget ElevatedButton puis appuyer sur Ctrl+Shift+R puis sélectionner Wrap with SizedBox.

Vous pouvez désormais ajouter votre paramètre height, sans oublier de créer une variable et de l'ajouter dans le constructeur.
Nous lui donnerons également une valeur par défaut au passage.

Ce qui nous donne finalement :

```
class CustomElevatedButton extends StatelessWidget {
  CustomElevatedButton({
    //...
    this.height = 50.0,
  });

  //...
  final double height;

  @override
  Widget build(BuildContext context) {
    return SizedBox(
      height: height,
      child: ElevatedButton(
        //...
      ),
    );
  }
}
```

Ajout des boutons restants

Nous avons désormais terminé avec la mise en place, nous pouvons désormais ajouter les boutons restants.

Nous allons donc créer un bouton pour la connexion avec Facebook, avec un email et anonyme.
Nous créerons également un simple texte de séparation entre avant le bouton de connexion anonyme.

De plus nous augmenterons la taille de l'espace sous le titre.

Avec les différents textes et codes couleurs, nous avons :

```
children: <Widget>[
  const Text(
    'Sign in',
    textAlign: TextAlign.center,
    style: TextStyle(fontSize: 32.0, fontWeight: FontWeight.w600),
  ), // Text
  const SizedBox(height: 48.0),
  SignInButton(
    text: 'Sign in with Google',
    textColor: Colors.black87,
    color: Colors.white,
    onPressed: () {},
  ), // SignInButton
  const SizedBox(height: 8.0),
  SignInButton(
    text: 'Sign in with Facebook',
    textColor: Colors.white,
    color: const Color(0xFF334D92),
    onPressed: () {},
  ), // SignInButton
  const SizedBox(height: 8.0),
  SignInButton(
    text: 'Sign in with Email',
    textColor: Colors.white,
    color: Colors.teal.shade700,
    onPressed: () {},
  ), // SignInButton
  const SizedBox(height: 8.0),
  const Text(
    'or',
    textAlign: TextAlign.center,
    style: TextStyle(fontSize: 14.0, color: Colors.black87),
  ), // Text
  const SizedBox(height: 8.0),
  SignInButton(
    text: 'Go anonymous',
    textColor: Colors.black87,
    color: Colors.lime.shade300,
    onPressed: () {},
  ), // SignInButton
], // <Widget>[]
```

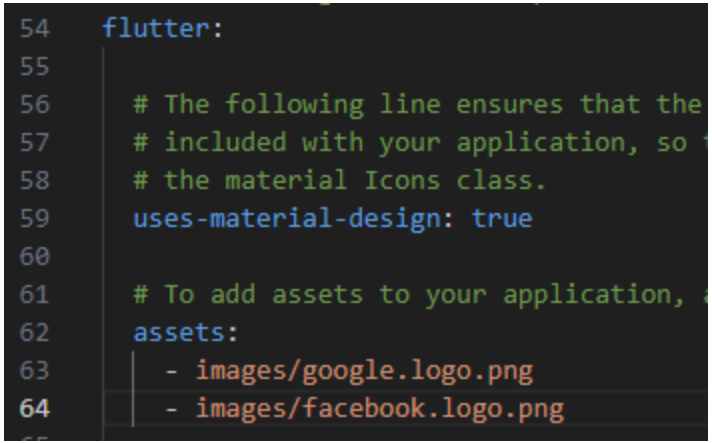
Ajouter les images dans le projet

Télécharger l'archive suivante localement : images

Extrayez ensuite les images dans un dossier images à la racine du projet.

Nous allons ensuite mettre à jour le fichier pubsec.yaml pour pouvoir utiliser nos images.

A la fin du fichier, vous devez apercevoir le mot flutter sans indentation, après la ligne concernant matérial design ajouter le code comme ci-dessous :



Comme préciser dans la documentation, il suffit d'indiquer le fichier de base pour sélectionner automatiquement toutes les versions différentes de l'images.

Ouvrez ensuite le terminal dans votre projet puis entrer la ligne de commande suivante :

```
flutter packages get
```

Ajout des logos

Pour ajouter une image, nous allons utiliser la méthode Image.asset pour importer notre image.

Pour le moment nous allons simplement reprendre notre CustomElevatedButton dans lequel nous allons remplacer notre image.

Nous nous retrouvons avec un bouton avec le code suivant :

```
CustomElevatedButton(
  child: Image.asset('images/google.logo.png'),
  color: Colors.white,
  onPressed: () {},
), // CustomElevatedButton
```

Ajout des logos

Puisque nous voulons également un texte sur la même ligne du bouton, nous allons utiliser le widget Row.

Nous utiliserons ensuite la paramètre ***mainAxisAlignment*** avec la valeur ***spaceBetween*** pour pouvoir rendre correctement notre image.

Notre image est bien placée mais le texte n'est pas centré. Pour résoudre le problème nous allons placer une deuxième image après le texte qui aura pour effet de le centrer.

```
CustomElevatedButton(
  color: Colors.white,
  onPressed: () {},
  child: Row(
    mainAxisAlignment: MainAxisAlignment.spaceBetween,
    children: [
      Image.asset('images/google.logo.png',
          width: 25.0, height: 25.0), // Image.asset
      const Text(
        'Sign in with Google',
        style: TextStyle(color: Colors.black87),
      ), // Text
      Image.asset('images/google.logo.png',
          width: 25.0, height: 25.0), // Image.asset
    ],
  ), // Row
), // CustomElevatedButton
```

SocialSignInButton

Comme précédemment, nous allons créer une nouvelle classe qui contiendra nos boutons avec des logos.

Pour cela créer un nouveau fichier ***social_sign_in_button.dart*** dans le ***dossier app.sign_in***.

Nous allons premièrement copier le code du fichier ***sign_in_button.dart*** dans notre fichier.
Nous copierons ensuite notre nouveau widget Row de notre prototype à la place de l'ancien widget Text de notre fichier.

Il ne reste plus qu'à remplacer les valeurs par les variables et de créer une nouvelle variable qui va contenir le chemin vers notre image.


```dart
import 'package:flutter/material.dart';
import 'package:myapp/common_widgets/custom_elevated_button.dart';

class SocialSignInButton extends CustomElevatedButton {
  SocialSignInButton({
    required String text,
    required Color textColor,
    required Color color,
    double borderRadius = 4.0,
    required VoidCallback onPressed,
    required String assetName,
  }) : super(
          child: Row(
            mainAxisAlignment: MainAxisAlignment.spaceBetween,
            children: <Widget>[
              Image.asset(
                assetName,
                width: 25.0,
                height: 25.0,
              ), // Image.asset
              Text(
                text,
                style: TextStyle(
                  color: textColor,
                  fontSize: 15.0,
                ), // TextStyle
              ), // Text
              Image.asset(
                assetName,
                width: 25.0,
                height: 25.0,
              ), // Image.asset
            ], // <Widget>[]
          ), // Row
          color: color,
          borderRadius: borderRadius,
          onPressed: onPressed,
        );
}
```

SocialSignInButton

Il ne reste plus qu'à importer notre nouvelle classe dans notre page est changer nos boutons ***Google*** et ***Facebook*** par notre nouveau bouton comme si dessous :

```
SocialSignInButton(
  assetName: 'images/google.logo.png',
  text: 'Sign in with Google',
  textColor: Colors.black87,
  color: Colors.white,
  onPressed: () {},
), // SocialSignInButton
```

```
SocialSignInButton(
  assetName: 'images/facebook.logo.png',
  text: 'Sign in with Facebook',
  textColor: Colors.white,
  color: const Color(0xFF334D92),
  onPressed: () {},
), // SocialSignInButton
```

SingleChildScrollView

SingleChildScrollView ne prend qu'un seul élément enfant, donc il ne peut faire défiler qu'un seul widget. C'est un widget de défilement, idéal pour des petits défilements. Cependant, pour un défilement nécessitant plus de ressources, on utilisera **ListView**.

```
return const Scaffold(
  appBar: MyAppBar(),
  body: SingleChildScrollView(
    child: Column(),
  ), // SingleChildScrollView
); // Scaffold
```

Formulaire et envoie de page

Pour créer les formulaires, nous allons concevoir un nouveau widget nommé SignInEmail.
Ce widget sera un StatefulWidget, car il subira des modifications.
Il comprendra un widget Form avec des champs de saisie, un bouton, et un traitement des informations.
Pour les champs de saisie et le widget Form :

```dart
class _SignInEmailState extends State<SignInEmail> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(
          title: const Text('Se connecter avec un email'),
        ), // AppBar
        body: SingleChildScrollView(
          padding: const EdgeInsets.all(16.0),
          child: Form(
            child: Column(
              mainAxisAlignment: MainAxisAlignment.center,
              children: [
                TextFormField(
                  decoration: const InputDecoration(
                    border: OutlineInputBorder(),
                    hintText: "Email",
                  ), // InputDecoration
                  validator: (value) {
                    if (value == null || value.isEmpty) {
                      return 'Veuillez rentrer un email';
                    }
                    return null;
                  },
                ), // TextFormField
```

Formulaire et envoie de page

```
            const SizedBox(
              height: 10,
            ), // SizedBox
            TextFormField(
              obscureText: true,
              decoration: const InputDecoration(
                border: OutlineInputBorder(),
                hintText: "Mot de passe",
              ), // InputDecoration
              validator: (value) {
                if (value == null || value.isEmpty) {
                  return 'Veuillez rentrer un mot de passe';
                }
                return null;
              },
            ), // TextFormField
            const SizedBox(
              height: 10,
            ), // SizedBox
            ElevatedButton(
              onPressed: () {},
              child: const Text('Se connecter'),
            ), // ElevatedButton
          ],
        ), // Column
      ), // Form
    )); // SingleChildScrollView // Scaffold
}
```

Dans le fichier, sign_in_page.dart, nous allons rajouter l'action lorsqu'on clique sue le bouton « Sign in with Email »

```dart
SignInButton(
  text: 'Sign in with Email',
  textColor: Colors.white,
  color: Colors.teal.shade700,
  onPressed: () {
    Navigator.push(
      context,
      MaterialPageRoute(builder: (context) => const SignInEmail()),
    );
  },
), // SignInButton
```

Formulaire et envoie de page

Après cela nous allons devoir créer des variables permettant de stocker les informations.
Tout d'abord une clé (propriété key dans notre widget Form()) qui va nous permettre de créer un conteneur pour les champs de notre formulaire.
Puis une variable pour chaque input qui va nous permettre de stocker les modifications des variables.

```
class _SignInEmailState extends State<SignInEmail> {
  final _formKey = GlobalKey<FormState>();
  final emailController = TextEditingController();
  final passwordController = TextEditingController();
  String email = '';
```

Formulaire et envoie de page

Nous modifions un minimum notre formulaire :

```
child: Form(
  key: _formKey,
  child: Column(
    mainAxisAlignment: MainAxisAlignment.center,
    children: [
      TextFormField(
        controller: emailController,
        decoration: const InputDecoration(
          border: OutlineInputBorder(),
          hintText: "Email",
        ), // InputDecoration
        validator: (value) {
          if (value == null || value.isEmpty) {
            return 'Veuillez rentrer un email';
          }
          return null;
        },
      ), // TextFormField
      const SizedBox(
        height: 10,
      ), // SizedBox
      TextFormField(
        controller: passwordController,
        obscureText: true,
```

Formulaire et envoie de page

Ensuite, créons la fonction signIn() qui validera notre formulaire :

```
void signIn() {
  if (_formKey.currentState!.validate()) {
    if (emailController.text == 'admin@admin.fr' &&
        passwordController.text == 'admin') {
      setState(() {
        email = emailController.text;
      });
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(
          content: Text("Bonjour, $email !"),
        ), // SnackBar
      );
    } else {
      ScaffoldMessenger.of(context).showSnackBar(
        const SnackBar(
          content: Text('Connexion échouée'),
        ), // SnackBar
      );
    }
  }
}
```

Formulaire et envoie de page

Appelons maintenant notre fonction dans le bouton de connexion.

```
ElevatedButton(
  onPressed: () {
    signIn();
  },
  child: const Text('Se connecter'),
), // ElevatedButton
```

Gestion des API

L'utilisation des Api est très simple en Flutter, vous allez avoir besoin:

- D'ajouter la dépendance http
- Créer une page de traitement
- Afficher dans une autre page

Gestion des API

Nous allons ajouter la dépendance http :

```
dart pub add http
```

C'est l'équivalent du "npm install" en nodeJs

Gestion des API

Ensuite créer une page pour récupérer les infos de notre api.

Pour l'exemple, nous utilisons l'api disponible en ligne de https://jsonplaceholder.typicode.com qui fournit une api en ligne pour effectuer nos tests.

```dart
import 'dart:convert';
import 'package:http/http.dart' as http; // Import http package

class Posts {
  // Variable contenant l'url de l'API
  static String baseUrl = "https://jsonplaceholder.typicode.com";

// Création d'une fonction permettant d'accéder à l'API
  static Future<List> getAllPosts() async {
    try {
      var response = await http.get(Uri.parse("$baseUrl/posts"));
      if (response.statusCode == 200) {
        return jsonDecode(response.body);
      } else {
        return Future.error("erreur serveur");
      }
    } catch (e) {
      return Future.error(e.toString());
    }
  }
}
```

Exemple pour l'affichage:

Vu que les variables sont possiblement modifiables en temps réel (si nous faisions un ajout, ou autre par exemple) nous allons donc utiliser un statefulWidget pour notre page ***anonymous_page.dart*** :

```dart
class _AnonymousPageState extends State<AnonymousPage> {
  late Future<List> _posts;

  @override
  void initState() {
    super.initState();
    _posts = Posts.getAllPosts();
  }
```

Exemple pour l'affichage:

Pour le retour de l'affichage, nous ajouterons une propriété future qui récupérera les informations de notre liste de posts.
Ensuite, dans le builder, nous vérifierons s'il y a des données et afficherons le résultat en conséquence :

```
Widget build(BuildContext context) {
  return Scaffold(
    body: FutureBuilder<List>(
        future: _posts,
        builder: (context, snapshot) {
          if (snapshot.hasData) {
            return ListView.builder(
                itemCount: snapshot.data!.length,
                itemBuilder: (context, index) {
                  return Card(
                      child: ListTile(
                    subtitle: Column(
                        crossAxisAlignment: CrossAxisAlignment.stretch,
                        children: [
                          Text(
                            snapshot.data![index]['title'],
                            style: const TextStyle(fontSize: 20),
                          ), // Text
                          Text(
                            snapshot.data![index]['body'],
                            style: const TextStyle(fontSize: 20),
                          ), // Text
                        ]), // Column
                  )); // ListTile // Card
                }); // ListView.builder
          } else {
            return const Center(child: Text("Pas de données !"));
          }
        }), // FutureBuilder
  ); // Scaffold
}
```

Dans le fichier *sign_in_page.dart*, nous allons ajouter l'action permettant d'afficher notre liste de posts lorsque nous cliquerons sur "*Go Anonymous*".

```
SignInButton(
  text: 'Go anonymous',
  textColor: Colors.black87,
  color: Colors.lime.shade300,
  onPressed: () {
    Navigator.push(
        context,
        MaterialPageRoute(
            builder: (context) => const AnonymousPage()));
  },
), // SignInButton
```

TP Call API

Vous allez pouvoir tester avec une de vos anciennes API, vous en avez réalisés plusieurs au cours de l'année, utilisez celle que vous souhaitez. La seule contrainte va être de réalisé un affichage, un ajout, suppression et une modification via l'appli flutter.