

# Flutter TD1 : Une première application

---

## Etape 1: Mise en place

Après avoir installé tous les outils nécessaires pour pouvoir faire du développement avec Flutter, nous allons pouvoir commencer à créer notre première application. L'objectif est de proposer aux utilisateur de répondre à un quizz très simple.

Lorsqu'on crée un projet Flutter, celui-ci contient par défaut une application dont le code se trouve dans le fichier *main.dart* du dossier *lib*. Nous étudierons éventuellement ce code ultérieurement mais pour ce TD, nous allons partir de "zéro". Donc, supprimons tout le contenu de *main.dart*.

Le dossier *test* du projet contient des fichiers qui permettent de tester l'application. Son contenu n'est donc pas adapté à notre future application. Il faut donc également supprimer tout le contenu du fichier *widget\_test.dart*.

En Dart, le point d'entrée d'un programme / d'une application est la fonction *main*. Testons le code suivant:

```
void main(){
  print("Hello World!!");
}
```

## Etape 2: Notre premier Widget

Pour construire une interface utilisateur, Flutter utilise des **widgets**. Tous les widgets sont consultables [ici](#). Une interface est construite à partir d'un widget racine auquel on va venir ajouter des widgets "enfants" et ainsi de suite. On parle donc souvent d'un arbre de widgets. La fonction *runApp()* qui prend en paramètre le widget racine permet de lancer l'application.

```
void main(){
  runApp(
    const Center(
      child: Text("Hello World !!!",textDirection: TextDirection.ltr),
    )
  );
}
```

## Etape 3: Un widget un peu plus complexe

Le widget de base que nous allons utiliser ici est *MaterialApp* (cf [doc](#)). C'est est une classe Flutter qui fournit une disposition de conception [material design](#) (langage de conception développé par Google). En fait, c'est un composant principal pour d'autres widgets enfants et on l'utilise généralement comme widget de niveau supérieur.

Chaque widget est paramétrable par ses attributs.

```

void main() {
  runApp(MaterialApp(
    debugShowCheckedModeBanner: false,
    title: "Application Quizz",
    home: Material( //Widget générique
      color: Colors.teal,
      shape:
        RoundedRectangleBorder(borderRadius:BorderRadius.circular(50.0) ),
      child: const Center(
        child: Text(
          "Hello world !!",
          textDirection: TextDirection.ltr,
          style: TextStyle(
            color: Colors.white,
            fontWeight: FontWeight.bold,
            fontStyle: FontStyle.italic,
            fontSize: 35.0,
          ),
        ),
      ),
    ),
  ));
}

```

## Etape 4: Créer ses propres widgets

Lorsqu'on développe une application, il est nécessaire de créer ses propres widgets afin de simplifier l'écriture du code, d'éviter du code redondant, .... Nous allons donc construire une classe qui hérite de *StatelessWidget*. Cette classe permet de créer un widget à partir d'un arbre de widgets. Pour cela, elle doit implémenter la méthode *build()* qui construit notre widget.

Créons un dossier *UI* dans *lib* dans lequel nous allons stocker nos widgets. Dans un fichier *home.dart*, écrivons le code suivant:

```

class MyWidget extends StatelessWidget{
  final Color color;
  final double textsize;
  final String message;
  const MyWidget(this.color,this.textsize,this.message);

  @override
  Widget build(BuildContext context) {
    return Material(
      color: color,
      shape:
        RoundedRectangleBorder(borderRadius:BorderRadius.circular(50.0) ),
      child: Center(
        child: Text(
          message,
          textDirection: TextDirection.ltr,
          style: TextStyle(
            color: Colors.white,
            fontWeight: FontWeight.bold,

```

```

        fontStyle: FontStyle.italic,
        fontSize: textsize,
    ),
    ));
}
}

```

La fonction *main()*:

```

void main() {
  runApp(const MaterialApp(
    debugShowCheckedModeBanner: false,
    title: "Application Quizz",
    home: MyWidget(Colors.teal,40.0,"Message du widget")
  ));
}

```

## Utilisation d'un Scaffold

Scaffold est une classe Flutter très importante qui implémente la structure de mise en page visuelle material design. Ce widget encapsule de nombreux widgets comme AppBar, Drawer, BottomNavigationBar, FloatingActionButton, SnackBar, etc (consultez la [doc](#)).

C'est un conteneur de niveau supérieur pour MaterialApp.

Dans un premier temps, nous allons utiliser simplement la *AppBar* qui possède également ses propres propriétés ([doc](#)). La méthode *build()* de la classe *MyWidget* devient:

```

Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: const Text("Quizz App"),
      centerTitle: true,
      backgroundColor: Colors.lightBlue,
    ),
    backgroundColor: color,
    body: Center(
      child: Text(
        message,
        textDirection: TextDirection.ltr,
        style: TextStyle(
          color: Colors.white,
          fontWeight: FontWeight.bold,
          fontStyle: FontStyle.italic,
          fontSize: textsize,
        ),
      ),
    ),
  ));
}

```

## Ajout d'une image

Chaque question de notre futur quizz sera accompagnée d'une image. Créons donc un dossier *images* dans notre projet afin de les stocker.

Afin que le Flutter soit capable de trouver les images du dossier, il faut ajouter le dossier *images* dans les *assets* du fichier *pubspec.yaml*.

```
assets:  
  - images/
```

Afin de placer une image au dessus du texte, utilisons le widget *Column* ([doc](#)). Afin d'espacer l'image et le texte, nous utilisons la propriété *mainAxisAlignment*.

```
Widget build(BuildContext context) {  
  return Scaffold(  
    appBar: AppBar(  
      title: const Text("Quizz App"),  
      centerTitle: true,  
      backgroundColor: Colors.lightBlue,  
    ),  
    backgroundColor: color,  
    body: Center(  
      child: Column(  
        mainAxisAlignment: MainAxisAlignment.spaceEvenly,  
        children: [  
          Image.asset("images/flag.png",width: 250,height: 180,),  
          Text(  
            ....  
          ),  
        ],  
      )),  
    ));
```

## Ajout d'une bordure

Pour ajouter une bordure autour du texte, utilisons le widget *Container* ([doc](#)). Ce widget peut donc avoir une bordure mais aussi des marges, une couleur de fond, des dimensions, ...

```
Widget build(BuildContext context) {  
  return Scaffold(  
    ...  
    body: Center(  
      child: Column(  
        mainAxisAlignment: MainAxisAlignment.spaceEvenly,  
        children: [  
          ...  
          Text(  
            ...  
          ),  
        ],  
      )),  
    ));
```

```

        Image.asset("images/flag.png",width: 250,height: 180,),
        Container(
          decoration: BoxDecoration(
            color: Colors.transparent,
            borderRadius: BorderRadius.circular(15),
            border: Border.all(
              color: Colors.black,style: BorderStyle.solid
            )
          ),
        ),
        height:120.0,
        child: Padding(
          padding: const EdgeInsets.all(20.0),
          child:Text(
            message,
            textDirection: TextDirection.ltr,
            style: TextStyle(
              ...
            ),
          )),],
      )),);
    }

```

## Ajout des boutons

Pour pouvoir répondre au quizz, nous allons ajouter quatre boutons alignés sur une ligne en dessous du texte pour répondre *Vrai*, *Faux* et passer à la question suivante et précédente.

Pour cela, il nous faut un widget *Row* et des *ElevatedButton* ([doc](#)).

Afin que chaque bouton ait le même style, nous allons commencer par créer un *ButtonStyle*.

```

final ButtonStyle myButtonStyle = ElevatedButton.styleFrom(
  backgroundColor: Colors.blueGrey.shade900,
  padding: const EdgeInsets.symmetric(horizontal: 20),
  shape: const RoundedRectangleBorder(
    borderRadius: BorderRadius.all(Radius.circular(20)),
  )
);

```

Ajoutons ensuite la ligne de quatre boutons à la liste des enfants du widget *Column*.

```

Row(
  mainAxisAlignment: MainAxisAlignment.spaceEvenly,
  children: [
    ElevatedButton(onPressed: ()=> _previousQuestion(),
      style: myButtonStyle,
      child: const Icon(Icons.arrow_back, color:
Colors.white,)),),
    ElevatedButton(onPressed: ()=> _checkAnswer(true),

```

```

        style: myButtonStyle,
        child: const Text("TRUE", style: TextStyle(color:
Colors.white))),),
        ElevatedButton(onPressed: ()=> _checkAnswer(false),
        style: myButtonStyle,
        child: const Text("FALSE", style: TextStyle(color:
Colors.white))),),
        ElevatedButton(onPressed: ()=> _nextQuestion(),
        style: myButtonStyle,
        child: const Icon(Icons.arrow_forward, color:
Colors.white),),),
    ]]]));

```

Pour le moment, les méthodes `_previousQuestion`, `_nextQuestion` et `_checkAnswer` ont un code vide. Nous verrons cela un peu plus tard.

## Mise ne place du modèle

Pour le fonctionnement de notre quizz, nous avons besoin d'un modèle de question. Une question est composée d'un intitulé, d'une réponse (vrai ou faux) et d'une image. Ajoutons un dossier *modele* à notre application dans lequel on crée un fichier *question.dart* avec le code suivant:

```

class Question{
  final String _questionText;
  final bool _isCorrect;
  final String _image;

  Question.name(this._questionText, this._isCorrect, this._image);

  bool get isCorrect => _isCorrect;

  String get questionText => _questionText;

  String get image => _image;
}

```


Nous pouvons maintenant créer la liste des questions qui va constituer notre quizz ainsi qu'une propriété indiquant la question en cours dans notre widget.

```

int _currentQuestion=0;

final List _questions = [Question.name("The question number 1 is a very
long question and her answer is true.", true, "flag.png"),
  Question.name("The question number 2 is true again.", true,
"img.png"),
  Question.name("The question number 3 is false.", false, "img.png"),
  Question.name("The question number 4 is false again.", false,
"flag.png"),
  Question.name("The question number 5 is true.", true, "flag.png"),

```

```
Question.name("The question number 6 is true again.", true,
, );
```

## Question facile

Modifiez le code de votre widget afin que le texte de la première question soit affiché et corrigez/modifiez votre arbre de widgets afin que l'affichage soit correct (pensez à tester avec un text assez long).

## Interactivité

On souhaiterait maintenant pouvoir répondre aux questions, passer à la question suivante ou précédente. Malheureusement, un *StatelessWidget* ne peut pas changer après avoir été dessiné. Flutter propose donc les *StatefulWidget* ([doc](#)) qui sont des widgets qui ont un état mutable.

L'état est une information qui (1) peut être lue de manière synchrone lorsque le widget est construit et (2) peut changer pendant la durée de vie du widget. Il est de la responsabilité de l'implémenteur du widget de s'assurer que l'État est rapidement informé lorsqu'un tel état change, en utilisant `State.setState`.

Pour implémenter un *StatefulWidget*, deux classes sont nécessaires:

- une qui hérite de la classe *StatefulWidget* qui définit le widget. Elle doit redéfinir la méthode `createState()` qui est appelée à la création du widget.
- l'autre qui hérite de la classe *State*. Elle contient l'état du widget ainsi que les données susceptibles de changer pendant la durée de vie du widget et elle implémente la méthode `build`.

Pour transformer un *StatelessWidget* en *StatefulWidget*, Android Studio propose le raccourci clavier `Alt+Entree`. On obtient:

```
class MyWidget extends StatefulWidget {
  final Color color;
  final double myTextSize;

  const MyWidget(this.color, this.myTextSize);

  @override
  State<MyWidget> createState() => _MyWidgetState();
}

class _MyWidgetState extends State<MyWidget> {
  final int _currentQuestion = 0;

  final List _questions = [...];

  @override
  Widget build(BuildContext context) {
    ...
  }

  _previousQuestion() {}
}
```

```

    _nextQuestion() {}

    _checkAnswer(bool rep) {}
}

```

Les méthodes `_previousQuestion()` et `_nextQuestion()` doivent changer le texte de la question. Il y a donc un changement d'état du widget. Ces méthodes doivent donc appeler la méthode `setState()` en indiquant les changements à effectuer sur les données pour demander au widget de se redessiner. Dans notre cas, il faut incrémenter ou décrémenter `_currentQuestion`.

```

_previousQuestion() {
  setState(() {
    _currentQuestion = (_currentQuestion-1)%_questions.length;
  });
}

_nextQuestion() {
  setState(() {
    _currentQuestion = (_currentQuestion+1)%_questions.length;
  });
}

```

Pour indiquer à l'utilisateur si il a bien répondu à la question, nous allons utiliser la `SnackBar` ([doc](#)). Dans ce cas, le widget n'est pas redessiné.

```

_checkAnswer(bool choice, BuildContext context) {
  if (choice == _questions[_currentQuestion].isCorrect){
    debugPrint("good");
    const mySnackBar = SnackBar(
      content: Text("GOOD ANSWER!!!",style: TextStyle(fontSize: 20)),
      duration: Duration(milliseconds: 500),
      backgroundColor: Colors.lightGreen,
      width: 180.0, // Width of the SnackBar.
      padding: EdgeInsets.symmetric(
        horizontal: 8.0, // Inner padding for SnackBar content.
      ),
      behavior: SnackBarBehavior.floating,
      shape: RoundedRectangleBorder(
        borderRadius: BorderRadius.all(Radius.circular(20)),
      ),
    );
    ScaffoldMessenger.of(context).showSnackBar(mySnackBar);
  }else{
    debugPrint("bad");
    const mySnackBar = SnackBar(
      content: Text("BAD ANSWER!!!",style: TextStyle(fontSize: 20)),
      duration: Duration(milliseconds: 500),
      backgroundColor: Colors.red,
      width: 180.0, // Width of the SnackBar.
    );
  }
}

```



```

padding: EdgeInsets.symmetric(
  horizontal: 8.0, // Inner padding for SnackBar content.
),
behavior: SnackBarBehavior.floating,
shape: RoundedRectangleBorder(
  borderRadius: BorderRadius.all(Radius.circular(20)),
),);
ScaffoldMessenger.of(context).showSnackBar(mySnackBar);
}
_nextQuestion();
}

```

## Un peu de refactorisation de notre code

Ici, l'objectif est de construire nos propres widgets pour obtenir les boutons. Nous allons donc construire deux classes: *MyTextButton* et *MyIconButton*.

### Mise en place d'un thème

Avant de commencer, déplaçons le style de nos boutons au niveau de l'application.

```

void main() {
  runApp(MaterialApp(
    theme: ThemeData(
      elevatedButtonTheme: ElevatedButtonThemeData(
        style: ElevatedButton.styleFrom(
          backgroundColor: Colors.blueGrey.shade900,
          padding: const EdgeInsets.symmetric(horizontal: 20),
          shape: const RoundedRectangleBorder(
            borderRadius: BorderRadius.all(Radius.circular(20)),
          ))
      )),
    debugShowCheckedModeBanner: false,
    title: "Application Quizz",
    home: const
MyWidget(Colors.teal,40.0)//MyWidget(Colors.teal,40.0,"Message super super
super super super long du widget")
));
}

```

### La classe *MyTextButton*

Ces boutons n'évoluent pas donc la classe hérite de *StatelessWidget*. Pour construire un bouton "texte", nous avons besoin du texte du bouton et de la valeur associée (*true* ou *false*). Mais il faut également informer le widget parent qu'il y a eu un click afin qu'il affiche la *snackbar*. Pour cela, le widget parent peut par exemple fournir un *callback* aux boutons via un attribut supplémentaire de type *ValueChanged* ([doc](#)). Dans un fichier *myButtons.dart* du dossier *UI*, ajoutons donc le code suivant:

```

class MyTextButton extends StatelessWidget{
  String myText;
  bool myValue;
  ValueChanged<bool> returnValue;

  MyTextButton({required this.myText, required this.myValue, required
  this.returnValue});

  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: () => returnValue(myValue),
      child: Text(myText,
        style: const TextStyle(color: Colors.white)),
    );
  }
}

```

Cette classe peut maintenant être utilisée pour remplacer les *ElevatedButtons* *TRUE* et *FALSE*.

```

MyTextButton(
  myText: "TRUE", myValue: true, returnValue: _handleValue),

MyTextButton(
  myText: "FALSE", myValue: false, returnValue: _handleValue),

```

La méthode *\_handleValue* correspond au *callback* appelé lors du click:

```

void _handleValue(bool value) {
  debugPrint(value.toString());
  if (value == _questions[_currentQuestion].isCorrect) {
    ...
    ScaffoldMessenger.of(context).showSnackBar(mySnackBar);
  } else {
    ...
    ScaffoldMessenger.of(context).showSnackBar(mySnackBar);
  }
  _nextQuestion();
}

```

Pour afficher la *snackbar*, il faut le *context* qu'on aura sauvegardé dans un attribut de classe afin de pouvoir y accéder dans cette méthode.

### La classe *MyIconButton*

L'utilisation d'un *callback* peut être compliquée quand le widget qui doit être informé n'est pas forcément le parent direct. Flutter propose donc beaucoup de classes permettant d'informer les widgets de l'arbre qui en

ont besoin qu'un changement a eu lieu. Ici, les boutons *précédent* et *suivant* provoquent un changement d'indice de la question courante. Le widget qui gère l'affichage de la question doit donc être informé afin de se redessiner avec la bonne question.

Dans cet exemple, nous allons utiliser la classe *Notification* ([doc](#)).

```
class IndexChanged extends Notification{
    final int val;

    IndexChanged(this.val);
}
```

La création d'une notification *IndexChanged* va permettre de fournir ici la valeur stockée dans *val* (1 ou -1 si on veut passer à la question suivante ou précédente) aux widgets qui attendent ces notifications via la méthode *dispatch*.

On peut donc construire notre classe *MyIconButton*

```
class MyIconButton extends StatelessWidget{
    IconData myIcon;
    int value;

    MyIconButton({required this.myIcon, required this.value});

    @override
    Widget build(BuildContext context) {
        return ElevatedButton(
            onPressed: () => IndexChanged(value).dispatch(context),

            child: Icon(
                myIcon,
                color: Colors.white,
            ),
        );
    }
}
```

Utilisons cette classe pour remplacer nos boutons:

```
MyIconButton(myIcon: Icons.arrow_back, value: -1),
MyTextButton(
    myText: "TRUE", myValue: true, returnValue: _handleValue),
MyTextButton(
    myText: "FALSE", myValue: false, returnValue: _handleValue),
MyIconButton(myIcon: Icons.arrow_forward, value: 1),
```

Le *body* du widget principal doit être informé des notifications et réagir lorsque cela se produit:

```
body:
    NotificationListener<IndexChanged>(
        child: Center(
            ...
        )
        onNotification: (n){
            _changeQuestion(n.val);
            return true;
        },
    )
```

et

```
_changeQuestion(int n){
    setState(() {
        _currentQuestion = (_currentQuestion + n) %_questions.length;
    });
}
```