

Développement Mobile

M2 DFS – 2025 / 2026

Plan de cours

01

Projet

Présentation de
l'application à réaliser

02

Revue des bases

Widgets, layouts et navigation
Ressources de développement

03

Interactions et gestion

Formulaires
Gestion des APIs

04

Design, animation, optimisation

Thèmes et personnalisation
Animation
Performances

00

Introduction

Tour de rappel rapide de Flutter et son fonctionnement

Flutter



Framework open-source développé par **Google** pour des applications mobiles, web et desktop avec une base code **unique** (langage Dart).

01



Multiplateforme

02



Performances

03



Hot Reload

04



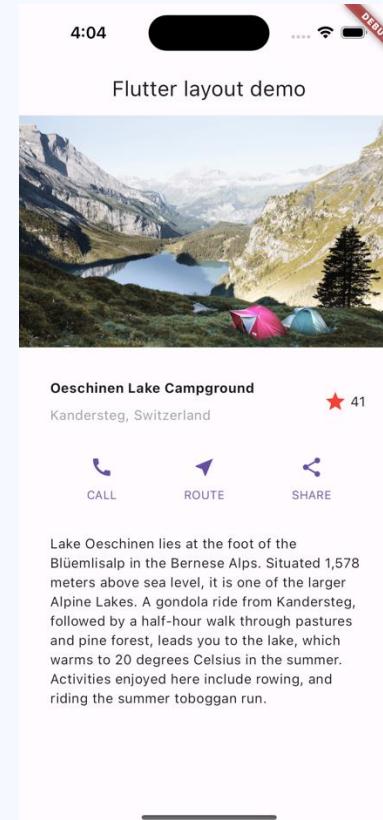
UI flexible

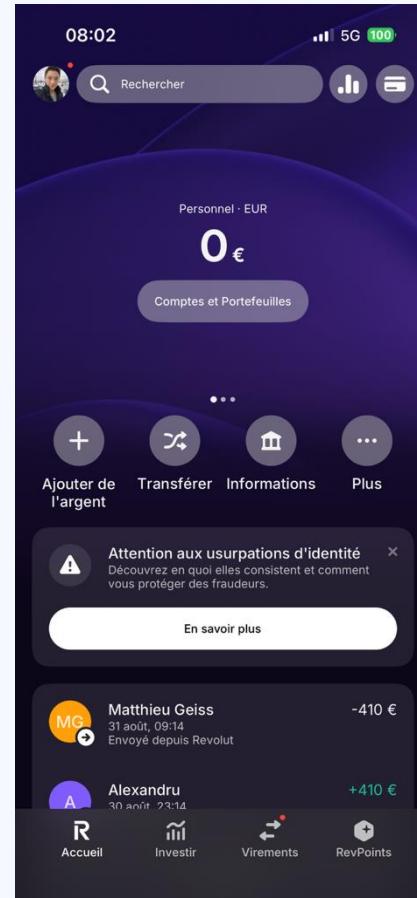
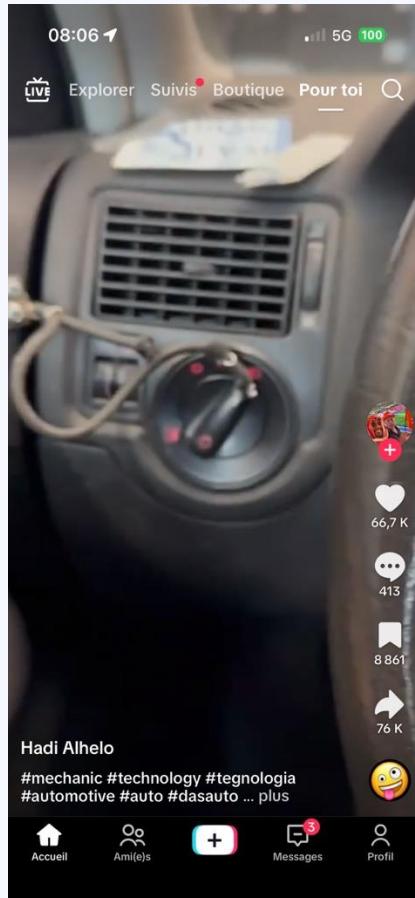
Fonctionnement des UIs

Définition des "widgets"?

Éléments de base de l'UI tels des blocs de construction organisés dans un arbre hiérarchique.

Dans Flutter tout est un widget.





08:05

5G 100%

← Q lego star wars

Sponsorié ⚠️

LEGO

One Piece Le Bateau Pirate
Vogue Merry - Jouet Bateau -
Maquette pour Décoration ave...

4,8 ★★★★★ (47)



109,99€ Conseillé : +29,99€
ou 27,50€/ 4 paiements (sans intérêts ni frais financiers)

Livraison GRATUITE jeu. 18 sept.

Âges : 10 ans et plus

Ajouter au panier

Sponsorié ⚠️

Star Wars

Micro Galaxy Squadron at-at
Walker Vaisseau Class Assault de
25 cm avec Cinq Micro Figurin...

4,6 ★★★★★ (331)



64,28€
ou 16,07€/ 4 paiements (sans intérêts ni frais financiers)

Livraison GRATUITE jeu. 18 sept.

Âges : 6 ans et plus

Ajouter au panier

Home Personne Panier Menu

08:03

5G 100%

For Amalia

Shows Movies Categories



N SERIES
BEAUTY AND THE BESTEST
Watch the Limited Series Now

▶ Play + My List

Continue Watching for Amalia



Home New & Hot My Netflix

Stateless / Stateful



Stateless

Le widget est **immutable**
et une fois créé il ne peut
pas changer

Ex: Texte statique, icônes



Stateful

Le widget est **mutable**
peut changer son état
interne

Ex: Barre de progression



pub.dev

Site officiel de gestion des packages pour Flutter et Dart

Commande d'installation d'un package:

`flutter pub add <package_name>`

01

Projet

Présentation de l'application mobile à réaliser



Piction.ia.ry

Règles du jeu

Le jeu est conçu pour 4 joueurs répartis dans 2 équipes de 2 joueurs avec pour rôles "**Dessinateur**" et "**Devineur**" qui seront inversés à chaque tour. Chaque duo commence avec un score de 100 points.

Préparation

Rédaction des challenges

Chaque joueur écrit **4 challenges** sous la forme

"Un/Une" [INPUT 1] "Sur/Dans Un/une" [INPUT 2]

Ainsi qu'une liste de **3 mots interdits** et les envoie à l'équipe adverse

Phase de jeu (5 minutes)

- 01** Le dessinateur reçoit le premier challenge. Il écrit un prompt qui sera envoyé à StableDiffusion qui ne peut pas comporter les mots à deviner ni les mots de la liste d'interdits
- 02** Le dessinateur envoie l'image au devineur. Il a la possibilité de regénérer l'image jusqu'au 2 fois pour un coût de 10 points
- 03** Le devineur tente de résoudre le challenge. Chaque proposition erronée coûte 1 point. Chaque mot trouvé rapporte 25 point
- 04** Lorsque le devineur a résolu son challenge, les rôles sont inversés
- 05** La partie s'arrête lorsque les équipes ont terminé leurs challenges ou si le temps est écoulé

Modalité d'évaluation

Phase 1 (10)

Tous les écrans (2)	- Tous les écrans sont designés
Design et convivialité (3)	<ul style="list-style-type: none">- Thème défini et utilisé (2)- Interface intuitive et facile à naviguer (1)
Phase de préparation (5)	<ul style="list-style-type: none">- Modèles (1)- Démarrage et lancement du jeu (2)- Envoi et réception des challenges à l'API (2)

Phase 2 (10)

App. fonctionnelle (4)	Le jeu est utilisable du début à la fin sans bug
Transition des écrans (2)	Gestion du navigateur et enchainement des écrans
Gestion du jeu (4)	<ul style="list-style-type: none">- Gestion des process en arrière plan asynchrones- Respect des règles- Gestion des actions- Mise à jour de la UI

UI/UX



Figma pour aide et inspiration



Thème

Définir le thème de votre application (couleurs, polices etc.)



Premier écran

Créer le premier écran avec les widgets adaptés



Personnalisation

Parcourez pub.dev à la recherche de package pour rendre agréable votre app.

build() / setState()

build()

- Crée l'interface d'un widget à chaque fois que Flutter doit redessiner
- Retourne un arbre de widget à afficher
- Appelée à chaque mise à jour de la UI
- Ne modifie pas les widget, seulement une recréation de l'arbre

```
Widget build2(BuildContext context) {  
  return Scaffold(  
    appBar: AppBar(title: Text('Mon App')),  
    body: Center(child: Text('Hello, Flutter!')),  
  ); // Scaffold  
}
```

setState()

- Informe Flutter qu'un état a changé provoquant la reconstruction du widget via build()
- Utilisé pour gérer l'interactivité et les modifications d'état
- Ne doit contenir que des modifications d'état, pas de logique de mise en page

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

Formulaires

Form()

- Widget permettant de regrouper plusieurs champs de saisie
- Peut valider l'ensemble des champs via des validators
- Identifié de manière unique par une clef

```
final GlobalKey<FormState> formKey = GlobalKey<FormState>();  
  
Form(  
    key: formKey, // Associe une clé au formulaire  
    child: Column(  
        children: [  
            TextFormField(  
                decoration: const InputDecoration(labelText: 'Nom'),  
                validator: (value) {  
                    if (value == null || value.isEmpty) {  
                        return 'Veuillez entrer un nom';  
                    }  
                    return null;  
                },  
            ), // TextFormField  
            ElevatedButton(  
                onPressed: () {  
                    if (formKey.currentState!.validate()) {  
                        // Soumission si le formulaire est valide  
                    }  
                },  
                child: const Text('Soumettre'),  
            ), // ElevatedButton  
        ],  
    ), // Column  
>; // Form
```

Controller (1/2)

- Gère l'état en stockant et manipulant les données liées au widget
- Permet l'accès direct au donner
- Ecoute les changement d'un widget
- Utilisé pour une interaction plus simple lors de gestion parfois complexe

```
@Override
void dispose() {
    _controller.dispose(); // Libération du controller
    super.dispose();
}

void myFunction(){
    TextField(
        controller: _controller,
        decoration: const InputDecoration(labelText: 'Saisissez du texte'),
    ); // TextField
}
```

Controller (2/2)

- Gère l'état en stockant et manipulant les données liées au widget
- Permet l'accès direct au donneur
- Ecoute les changements d'un widget
- Utilisé pour une interaction plus simple lors de gestion parfois complexe

```
void _scrollToTop() {
    _scrollController.animateTo(
        0.0, // Position 0 (en haut de la liste)
        duration: Duration(seconds: 1), // Durée de l'animation
        curve: Curves.easeInOut, // Courbe de l'animation
    );
}

@Override
void dispose() {
    _scrollController.dispose(); // Libérer le ScrollController
    super.dispose();
}

@Override
Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(
            title: const Text('Scroll to Top Example'),
        ), // AppBar
        body: ListView.builder(
            controller: _scrollController, // Attache le ScrollController
            itemCount: 50,
            itemBuilder: (context, index) {
                return ListTile(
                    title: Text('Item $index'),
                ); // ListTile
            },
        ), // ListView.builder
        floatingActionButton: FloatingActionButton(
            onPressed: _scrollToTop, // Action pour remonter en haut
            child: const Icon(Icons.arrow_upward), // Icône de flèche vers le haut
        ), // FloatingActionButton
    ); // Scaffold
}
```

Navigator (routes et push)

Les routes

- Représente un écran ou une page dans l'application
- Flutter utilise une pile (stack) pour gérer les routes, le dernier écran empilé est celui visible

```
MaterialApp(  
  routes: {  
    '/': (context) => const HomeScreen(),  
    '/second': (context) => const SecondScreen(),  
  },  
); // MaterialApp
```

Navigation

- Navigator.push() : Ajoute une nouvelle route à la pile (pushReplacement = remplace)
- Navigator.pop() : Retire la route actuelle de la pile

```
Navigator.push(  
  context,  
  MaterialPageRoute(builder: (context) => const SecondScreen()),  
);
```

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

```
TextButton(  
  onPressed: () {  
    Navigator.pop(context);  
  },  
  child: const Text('Retour'),  
); // TextButton
```

Asynchronisme

Concept de l'asynchronisme

Permet d'exécuter des opérations sans bloquer le fil principal d'exécution (UI)

Mot-clés

- **Future**: Représente une valeur qui peut être disponible à l'avenir
- **async**: Indique qu'une fonction contient des opérations asynchrones
- **await**: Attend la complétion d'un Future avant de continuer l'exécution (bloquant)

Conseil

Utiliser try/catch pour gérer les exceptions lors des opérations asynchrones.

```
Future<String> fetchData() async {
    await Future.delayed(Duration(seconds: 2)); // Simule un délai
    return 'Données récupérées';
}

void getData() async {
    String data = await fetchData(); // Attend la complétion
    debugPrint(data); // Affiche "Données récupérées"
}
```

HTTP

<https://pub.dev/packages/http>

Utilité

Plugin permettant de faire des requêtes HTTP vers une adresse web (par exemple une API)

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

Future<Map<String, dynamic>> fetchData() async {
  var url = Uri.https('api.com', 'collection/create');
  var response = await http.post(url, body: {'name': 'Matthieu', 'color': 'yellow'});
  debugPrint('Response status: ${response.statusCode}');
  debugPrint('Response body: ${response.body}');

  return jsonDecode(response.body);
}
```

```
Future<void> postData() async {
  final response = await http.get(Uri.parse('https://jsonplaceholder.typicode.com/posts/1'));
  if (response.statusCode == 200) {
    debugPrint('Données récupérées : ${response.body}');
  } else {
    debugPrint('Erreur : ${response.statusCode}');
  }
}
```

```
Future<void> postData() async {
  final response = await http.post(
    Uri.parse('https://jsonplaceholder.typicode.com/posts'),
    body: {'title': 'foo', 'body': 'bar', 'userId': '1'},
  );
  debugPrint('Réponse : ${response.body}');
}
```

FutureBuilder

Utilité

Permet de construire une interface en fonction de l'état d'un Future

Il permet d'afficher les données assynchrones une fois le future complété

Etats du FutureBuilder

ConnectionState.waiting

Le Future est en attente (on affiche souvent un indicateur)

ConnectionState.done

Le Future est complété (Affichage des données ou d'une erreur)

```
Future<String> fetchData() async {
  await Future.delayed(const Duration(seconds: 2));
  return 'Données récupérées';
}

@Override
Widget build(BuildContext context) {
  return FutureBuilder<String>(
    future: fetchData(),
    builder: (context, snapshot) {
      if (snapshot.connectionState == ConnectionState.waiting) {
        return const CircularProgressIndicator(); // Affichage pendant le chargement
      } else if (snapshot.hasError) {
        return Text('Erreur : ${snapshot.error}');
      } else {
        return Text('Résultat : ${snapshot.data}');
      }
    },
  ); // FutureBuilder
}
```

CachedNetworkImage

https://pub.dev/packages/cached_network_image

Concept et utilité

Permet de charger puis d'afficher une image depuis une adresse externe et de la mettre en cache.

Si le réseau n'est pas disponible ou si une version en cache de l'image est disponible alors le widget ira automatiquement chercher la version locale du fichier

```
Widget build(BuildContext context) {
  return CachedNetworkImage(
    imageUrl: "http://via.placeholder.com/350x150",
    placeholder: (context, url) => const CircularProgressIndicator(),
    errorWidget: (context, url, error) => const Icon(Icons.error),
  ); // CachedNetworkImage
}

Widget build2(BuildContext context) {
  return CachedNetworkImage(
    imageUrl: "http://via.placeholder.com/350x150",
    progressIndicatorBuilder: (context, url, downloadProgress) =>
      CircularProgressIndicator(value: downloadProgress.progress),
    errorWidget: (context, url, error) => const Icon(Icons.error),
  ); // CachedNetworkImage
}
```

Bibliothèque avec alias

Concept et utilité

Permet d'importer un fichier (ou un package) tout en lui attribuant un alias pour éviter les conflits de nommage et faciliter la lecture du code.

Offre la possibilité d'accéder à des données globales partagées comme des données de sessions.

```
// data.dart  
String token = "";
```

```
import 'package:app/data.dart' as data;  
  
void main() {  
    // Attribution de la variable token via l'alias "data"  
    data.token = "xxx";  
  
    print(data.token); // Affiche "xxx"  
}
```

SharedPreferences

https://pub.dev/packages/shared_preferences

Concept et utilité

Utilisé pour stocker des **données légères** de manière persistante

- Stocke des **données simples** sous la forme clé-valeur localement, de manière persistante.

- Idéal pour sauvegarder des préférences utilisateur (ex : thème, connexion automatique, etc.).

```
Future<void> prefs() async {  
  SharedPreferences prefs = await SharedPreferences.getInstance();  
  
  prefs.setBool('isLoggedIn', true);  
  bool? isLoggedIn = prefs.getBool('isLoggedIn');  
}
```

Méthodes principales :

`setX()` : Sauvegarder une valeur (ex: `setBool()`, `setString()`)

`getX()` : Récupérer une valeur sauvegardée

`remove()` : Supprimer une clé spécifique

Hero

Concept et utilité

Utilisé pour créer des transitions animées entre deux écrans

Idéal pour des effets de continuité visuelle

Fonctionnement

Identifiant unique (tag) partagé entre les deux écrans

Animation automatique par Flutter

```
Hero(  
  tag: 'profilePic', // Identifiant unique partagé  
  child: Image.asset('assets/profile.jpg'),  
) // Hero
```

```
Widget build(BuildContext context) {  
  return Scaffold(  
    appBar: AppBar(  
      title: const Text('Home'),  
    ), // AppBar  
    body: Center(  
      child: Column(  
        children: [  
          const Hero(  
            tag: 'profile',  
            child: FlutterLogo(size: 100),  
          ), // Hero  
          ElevatedButton(  
            onPressed: () {  
              Navigator.of(context).push(MaterialPageRoute<void>(  
                builder: (BuildContext context) {  
                  return const ProfilScreen();  
                },  
              )); // MaterialPageRoute  
            },  
            child: const Text('Go to Home'),  
          ), // ElevatedButton  
        ],  
      ), // Column  
    ), // Center  
  ); // Scaffold
```

```
Widget build(BuildContext context) {  
  return Scaffold(  
    appBar: AppBar(  
      title: const Text('Profile'),  
    ), // AppBar  
    body: const Column(  
      children: [  
        Hero(  
          tag: 'profile',  
          child: FlutterLogo(size: 300),  
        ), // Hero  
        Text('Profile Screen'),  
      ],  
    ), // Column  
  ); // Scaffold
```

Staggered Animations

https://pub.dev/packages/flutter_staggered_animations

Concept et utilité

Bibliothèque Flutter qui permet de créer des animations séquentielles ou "échelonnées" facilement.

Amélioration de l'UX

Permet de rendre l'interface plus fluide en animant une grille, une liste ou même l'écran affiché

```
AnimationConfiguration.staggeredList(  
  position: index,  
  child: SlideAnimation(  
    child: FadeInAnimation(  
      child: ListTile(  
        title: Text('Item $index'),  
      ), // ListTile  
    ), // FadeInAnimation  
  ), // SlideAnimation  
); // AnimationConfiguration.staggeredList
```

```
Widget animateThis({required Widget child, required int i}) {  
  return AnimationConfiguration.staggeredList(  
    position: i,  
    duration: const Duration(milliseconds: 500),  
    child: SlideAnimation(  
      verticalOffset: 50.0,  
      child: FadeInAnimation(child: child),  
    ), // SlideAnimation  
  ); // AnimationConfiguration.staggeredList  
}
```

Timer.periodic

Concept et utilité

Permet de lancer de façon cyclique plusieurs opérations

Recommandations

Stopper le timer si vous n'en avez plus besoin

Gérer les mises en veilles

```
void startRepeatingTask() {  
    // Exécute la tâche toutes les 5 secondes  
    Timer.periodic(const Duration(seconds: 5), (Timer timer) {  
        // Code à exécuter périodiquement  
        debugPrint("Tâche répétitive exécutée !");  
    }); // Timer.periodic  
}
```

```
late Timer _timer;  
  
void startRepeatingTask() {  
    _timer = Timer.periodic(Duration(seconds: 5), (Timer timer) {  
        print("Tâche répétitive exécutée !");  
    }); // Timer.periodic  
}  
  
// Pour arrêter le Timer  
void stopRepeatingTask() {  
    if (_timer != null) {  
        _timer.cancel();  
    }  
}
```

Future.delayed

Concept et utilité

Permet de programmer une tâche dans le futur

```
void startDelayedTask() {  
  Future.delayed(const Duration(seconds: 10), () {  
    debugPrint("Tâche programmée exécutée après 10 secondes !");  
}); // Future.delayed  
}
```

Recommandations

Annuler l'action si vous n'en avez plus besoin avec un boolean dans le `dispose()` du widget

Gérer les cycles de vies du widget avec **initState** et **dispose**