Module J3 : Navigation et Mise en Page (Partie 2)

Objectif : Gérer la navigation entre les écrans et maîtriser les mises en page complexes.

3.1. Navigation Basique (Durée Estimée : 40 min)

La navigation dans une application mobile est essentielle. Dans Flutter, la navigation est gérée par le Widget Navigator, qui fonctionne sur le principe d'une **pile (Stack)**.

A. Le Concept de Pile d'Écrans

Imaginez que chaque écran (ou page) de votre application est une carte. Lorsque vous naviguez vers un nouvel écran, vous empilez une nouvelle carte sur le dessus. Lorsque vous revenez en arrière, vous retirez la carte du dessus.

- **Écran A** (en bas de la pile)
- **Écran B** (au-dessus de A)
- **Écran C** (au sommet de la pile - l'écran visible)

B. Les Méthodes Clés du Navigator

Le Navigator est accessible via Navigator.of(context).

1. push() : Ajouter un nouvel écran

La méthode push() ajoute un nouvel écran au sommet de la pile. Elle prend un Route comme argument, le plus souvent un MaterialPageRoute.

```
// Depuis l'Écran A, on navigue vers l'Écran B
```

```
ElevatedButton(
  child: const Text('Aller à l\'Écran B'),
  onPressed: () {
    Navigator.of(context).push(
      MaterialPageRoute(
        builder: (context) => const EcranB(), // On définit le Widget de
destination
      ),
    );
  },
```

```
)
```

2. pop() : Revenir à l'écran précédent

La méthode pop() retire l'écran actuel du sommet de la pile, révélant l'écran qui se trouve en dessous.

```
// Depuis l'Écran B, on revient à l'Écran A

ElevatedButton(
  child: const Text('Retour'),
  onPressed: () {
    Navigator.of(context).pop();
  },

)
```

C. Passage de Données entre les Écrans

Il est courant de devoir passer des données d'un écran à l'autre (par exemple, l'identifiant d'un produit).

1. Passer des données à l'écran de destination

Les données sont passées via le constructeur de l'écran de destination.

Écran de Destination (EcranDetail) :

```
class EcranDetail extends StatelessWidget {

  final String titreProduit; // La donnée à recevoir

  const EcranDetail({Key? key, required this.titreProduit}) : super(key:
key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text(titreProduit)),
      body: Center(child: Text('Détails pour : $titreProduit')),
    );
  }

}
```

Écran d'Origine (Navigation) :

```
// On passe la donnée lors de l'appel à push
```

```
ElevatedButton(
  child: const Text('Voir le Produit'),
  onPressed: () {
    Navigator.of(context).push(
      MaterialPageRoute(
        builder: (context) ⇒ const EcranDetail(titreProduit: 'Smartphone
X'),
      ),
    );
  },
```

```
)
```

2. Retourner des données à l'écran d'origine

L'écran de destination peut renvoyer un résultat à l'écran d'origine en utilisant la méthode pop() avec un argument.

Écran de Destination (Retour) :

```
// On renvoie une valeur (ici, un booléen) lors du pop
```

```
ElevatedButton(
  child: const Text('Confirmer et Retourner'),
  onPressed: () {
    Navigator.of(context).pop(true); // Renvoie 'true'
  },
```

```
)
```

Écran d'Origine (Réception) :

La méthode push() retourne un Future. On utilise async et await pour attendre le résultat.

```
ElevatedButton(
  child: const Text('Aller à l\'Écran de Choix'),
  onPressed: () async {
    final resultat = await Navigator.of(context).push(
      MaterialPageRoute(builder: (context) => const EcranChoix()),
    );

    if (resultat == true) {
      // Le résultat est reçu ici
      ScaffoldMessenger.of(context).showSnackBar(
        const SnackBar(content: Text('Choix confirmé !')),
      );
    }
  },
)
```

Exercice Pratique (3.1)

1. **Créer deux écrans :** EcranPrincipal et EcranSecondaire.
2. Dans EcranPrincipal, ajouter un bouton qui navigue vers EcranSecondaire.
3. Dans EcranSecondaire, ajouter un bouton qui revient à EcranPrincipal et renvoie la chaîne de caractères "Opération réussie".
4. Dans EcranPrincipal, afficher le message renvoyé par EcranSecondaire dans la console ou un SnackBar.

3.2. Mises en Page Avancées (Durée Estimée : 80 min)

A. Listes Efficaces : ListView

Le ListView est le Widget de base pour afficher une liste d'éléments défilants.

1. ListView Simple

Utilisé pour les listes courtes dont tous les éléments peuvent être chargés en mémoire sans problème.

```
ListView(
  children: const <Widget>[
    ListTile(title: Text('Élément 1')),
    ListTile(title: Text('Élément 2')),
    // ...
  ],
)
```

2. ListView.builder (Optimisation Clé)

C'est la méthode recommandée pour les listes longues ou potentiellement infinies. Elle construit les Widgets **à la demande** (uniquement ceux qui sont visibles à l'écran), ce qui est crucial pour la performance.

```
final List<String> elements = List.generate(1000, (i) => 'Article $i');

ListView.builder(
  itemCount: elements.length,
  itemBuilder: (context, index) {
    return ListTile(
      title: Text(elements[index]),
    );
  },
)
```

B. Grilles d'Éléments : GridView

Le GridView permet d'afficher des Widgets dans une grille bidimensionnelle défilante.

1. GridView.count

Utilisé pour spécifier un nombre fixe de colonnes.

```
GridView.count(
  crossAxisCount: 3, // Nombre de colonnes
  children: List.generate(9, (index) {
    return Center(
      child: Text('Case $index'),
    );
  }),
)
```

2. GridView.extent

Utilisé pour spécifier la taille maximale de chaque élément, laissant Flutter calculer le nombre de colonnes.

C. Widgets Superposés : Stack

Le Stack permet de superposer des Widgets les uns sur les autres, du bas vers le haut (le premier enfant est le plus bas, le dernier est le plus haut).

- **Utilisation :** Placer du texte sur une image, afficher un badge de notification sur une icône.
- **Widget Positioned :** Utilisé à l'intérieur d'un Stack pour positionner précisément un enfant (top, bottom, left, right).

```
Stack(
```

```
  children: <Widget>[
    // 1. L'image (en bas)
    Image.network('URL_IMAGE'),

    // 2. Le texte (au-dessus, positionné en bas à gauche)
    const Positioned(
      bottom: 10,
      left: 10,
      child: Text(
        'Titre de l\'image',
        style: TextStyle(color: Colors.white, fontSize: 24),
      ),
    ),
  ],
```

```
)
```

D. Introduction aux Sliver

Les Sliver sont des Widgets spécialisés utilisés pour créer des effets de défilement personnalisés et avancés.

- **CustomScrollView :** Le Widget qui doit être utilisé pour contenir des Sliver.
- **SliverAppBar :** Une barre d'application qui peut se réduire ou s'étendre lorsque l'utilisateur fait défiler l'écran (effet "Parallax").
- **SliverList / SliverGrid :** Versions Sliver de ListView et GridView.

Exemple d'utilisation :

```
CustomScrollView(
```

```
  slivers: <Widget>[
    const SliverAppBar(
      pinned: true, // La barre reste visible en haut
      expandedHeight: 250.0,
      flexibleSpace: FlexibleSpaceBar(
        title: Text('Mon Profil'),
      ),
    ),
    SliverList(
      delegate: SliverChildBuilderDelegate(
        (BuildContext context, int index) {
          return ListTile(title: Text('Élément $index'));
        },
        childCount: 100,
      ),
    ),
  ],
```

```
)
```

Exercice Pratique (3.2)

5 **Créer une liste de 100 éléments** à l'aide de ListView.builder pour garantir l'efficacité.

6 **Créer une galerie de 6 images** (simulées par des Container de couleurs différentes) en utilisant GridView.count.

7 **Utiliser un Stack** pour afficher un Container rouge de 50x50 pixels dans le coin supérieur droit d'une image (ou d'un autre Container plus grand).

Exercice Pratique (3.2)

8 **Créer une liste de 100 éléments** à l'aide de ListView.builder pour garantir l'efficacité.

9 **Créer une galerie de 6 images** (simulées par des Container de couleurs différentes) en utilisant GridView.count.

10 **Utiliser un Stack** pour afficher un Container rouge de 50x50 pixels dans le coin supérieur droit d'une image (ou d'un autre Container plus grand).