



# INTRODUCTION À FLUTTER

# PRÉSENTATION DE FLUTTER

Flutter est un framework open-source créé par Google. Il permet de développer des applications nativement compilées. Les applications peuvent être déployées sur mobile, web et desktop. Flutter utilise le langage de programmation Dart. Il offre une approche unique pour le développement d'UI. Flutter est connu pour sa rapidité et son efficacité. Il fournit un riche ensemble de widgets personnalisables. Les développeurs peuvent créer des interfaces utilisateur attrayantes. Flutter facilite le développement cross-platform. Il est soutenu par une grande communauté de développeurs.

# HISTORIQUE DE FLUTTER

Flutter a été annoncé par Google en 2015. La première version stable, Flutter 1.0, est sortie en décembre 2018. Il a rapidement gagné en popularité parmi les développeurs. Flutter 2.0 est sorti en mars 2021, ajoutant le support pour le web. Le framework continue de recevoir des mises à jour régulières. Google utilise Flutter pour plusieurs de ses propres applications. Flutter a été conçu pour rivaliser avec React Native. Il est devenu un choix populaire pour le développement cross-platform. La communauté Flutter s'est fortement développée depuis son lancement. Flutter est maintenant utilisé dans de nombreux secteurs d'activité.

# AVANTAGES DE FLUTTER

Développement cross-platform avec un seul codebase. Hot Reload pour voir les changements instantanément. Performance proche des applications natives. Riche ensemble de widgets personnalisables. Documentation complète et bien organisée. Soutien d'une grande communauté de développeurs. Intégration facile avec les services backend. Support pour le développement mobile, web et desktop. Facilité de test et de débogage. Partenariat avec des entreprises majeures comme Google.

# APPLICATIONS CÉLÈBRES UTILISANT FLUTTER

Google Ads Alibaba Reflectly Birch Finance Hamilton Musical Hookle Abbey Road Studios Philips Hue  
Tencent BMW

# ARCHITECTURE DE FLUTTER

Flutter utilise une architecture réactive. Le framework est divisé en trois couches principales :

- Framework : écrit en Dart, contient les widgets.
- Engine : écrit en C++, gère le rendu graphique.
- Embedder : spécifique à la plateforme, intègre Flutter à l'OS. Le moteur de rendu utilise Skia pour le dessin 2D. Les widgets sont construits de manière hiérarchique. Flutter utilise le concept de "composition over inheritance". Les animations et transitions sont fluides et performantes. Le framework gère les états et les mises à jour de l'UI. La communication entre Dart et le code natif est facile. Flutter permet une grande flexibilité et personnalisation.

# INSTALLATION DE FLUTTER

# TÉLÉCHARGEMENT DE FLUTTER SDK

1. Rendez-vous sur le site officiel de Flutter : [flutter.dev](https://flutter.dev)
2. Cliquez sur "Get Started".
3. Sélectionnez votre système d'exploitation (Windows, macOS, Linux).
4. Téléchargez le fichier zip du SDK Flutter.
5. Extrayez le contenu du fichier zip dans un répertoire de votre choix.

# INSTALLATION DES OUTILS DE LIGNE DE COMMANDE

# WINDOWS

1. Ajoutez Flutter à votre PATH :

- Ouvrez les paramètres système avancés.
- Cliquez sur "Variables d'environnement".
- Ajoutez le chemin du répertoire **flutter/bin** à la variable PATH.

# MACOS/LINUX

1. Ouvrez le terminal.
2. Ajoutez Flutter à votre PATH :
  - Ajoutez `export PATH="$PATH:[chemin_vers_flutter]/flutter/bin"` à votre fichier de profil (`.bashrc`, `.zshrc`, etc.).

# CONFIGURATION DES VARIABLES D'ENVIRONNEMENT

1. Ouvrez le terminal ou l'invite de commande.
2. Configurez les variables d'environnement nécessaires :
  - `PUB_HOSTED_URL=https://pub.flutter-io.cn`
  - `FLUTTER_STORAGE_BASE_URL=https://storage.flutter-io.cn`
3. Enregistrez les modifications et redémarrez le terminal.

# VÉRIFICATION DE L'INSTALLATION AVEC FLUTTER DOCTOR

1. Ouvrez le terminal ou l'invite de commande.
2. Exécutez la commande suivante :

```
flutter doctor
```

3. Vérifiez les résultats :
  - Flutter doit être correctement installé.
  - Les outils de développement Android/iOS doivent être configurés.
  - Les IDE (Android Studio, VS Code) doivent être détectés.

# CONFIGURATION DE L'ENVIRONNEMENT DE DÉVELOPPEMENT

# INSTALLATION DE L'IDE (ANDROID STUDIO, VS CODE)

1. Téléchargez Android Studio depuis le site officiel.
2. Installez Android Studio en suivant les instructions.
3. Téléchargez Visual Studio Code depuis le site officiel.
4. Installez Visual Studio Code en suivant les instructions.
5. Ouvrez Android Studio et configurez le SDK Android.
6. Ouvrez Visual Studio Code et installez les extensions Flutter et Dart.

# CONFIGURATION DES ÉMULATEURS ET SIMULATEURS

1. Ouvrez Android Studio.
2. Allez dans "AVD Manager" via le menu "Tools".
3. Créez un nouvel émulateur en suivant les instructions.
4. Sélectionnez un appareil virtuel et une image système.
5. Lancez l'émulateur créé.
6. Pour iOS, utilisez Xcode pour configurer les simulateurs.

# CONFIGURATION DES VARIABLES D'ENVIRONNEMENT

1. Ajoutez Flutter au PATH.

2. Sous Windows :

- Ouvrez "Paramètres système avancés".
- Cliquez sur "Variables d'environnement".
- Ajoutez le chemin de Flutter à la variable PATH.

3. Sous macOS et Linux :

- Ouvrez le fichier `.bashrc` ou `.zshrc`.
- Ajoutez `export PATH="$PATH:[chemin vers flutter]/bin"`.

# INSTALLATION DES PLUGINS NÉCESSAIRES

1. Ouvrez Visual Studio Code.
2. Allez dans l'onglet "Extensions".
3. Recherchez "Flutter" et installez-le.
4. Recherchez "Dart" et installez-le.
5. Ouvrez Android Studio.
6. Allez dans "Plugins" et installez les plugins Flutter et Dart.

# VÉRIFICATION DE L'INSTALLATION AVEC FLUTTER DOCTOR

1. Ouvrez un terminal.
2. Tapez la commande `flutter doctor`.
3. Vérifiez les résultats pour s'assurer qu'il n'y a pas d'erreurs.
4. Installez les composants manquants si nécessaire.
5. Assurez-vous que tous les outils sont correctement configurés.
6. Répétez la commande `flutter doctor` jusqu'à ce que tout soit en ordre.

# CRÉATION D'UN NOUVEAU PROJET FLUTTER

1. Ouvrez Visual Studio Code.
2. Allez dans le menu "View" > "Command Palette".
3. Tapez **Flutter: New Project**.
4. Choisissez un nom pour votre projet.
5. Sélectionnez l'emplacement de sauvegarde du projet.
6. Attendez que le projet soit créé.

# CONFIGURATION INITIALE DU PROJET (PUBSPEC.YAML, ETC.)

1. Ouvrez le fichier `pubspec.yaml`.
2. Ajoutez les dépendances nécessaires sous `dependencies`.
3. Exemple de dépendance :

```
dependencies:  
  flutter:  
    sdk: flutter  
  cupertino_icons: ^1.0.2
```

4. Sauvegardez le fichier.
5. Exécutez la commande `flutter pub get` pour récupérer les packages.
6. Configurez les autres fichiers de configuration si nécessaire.

# STRUCTURE DE BASE D'UNE APPLICATION FLUTTER

# NOTION DE PROJET FLUTTER

Un projet Flutter est structuré pour faciliter le développement d'applications mobiles. Il contient plusieurs répertoires et fichiers essentiels. Les principaux répertoires sont `lib`, `test`, `android`, `ios`, et `web`. Le répertoire `lib` est où vous écrivez la plupart de votre code Dart. Le fichier `pubspec.yaml` gère les dépendances et les configurations du projet. Le fichier `main.dart` est le point d'entrée de l'application.

# STRUCTURE DES RÉPERTOIRES

- **lib/**: Contient le code source Dart de l'application.
- **test/**: Contient les tests unitaires et d'intégration.
- **android/**: Contient le code spécifique à la plateforme Android.
- **ios/**: Contient le code spécifique à la plateforme iOS.
- **web/**: Contient le code spécifique à la plateforme Web.
- **build/**: Contient les fichiers générés lors de la compilation.

# FICHIERS PRINCIPAUX (MAIN.DART, PUBSPEC.YAML)

- `main.dart`: Point d'entrée de l'application Flutter.
- `pubspec.yaml`: Gère les dépendances, les assets, et les configurations de l'application.
- `README.md`: Documentation du projet.
- `build.gradle`: Fichier de configuration pour la compilation Android.
- `Info.plist`: Fichier de configuration pour iOS.

# FONCTION MAIN() ET RUNAPP()

La fonction `main()` est le point de départ de toute application Flutter. Elle appelle généralement la fonction `runApp()` pour lancer l'application.

```
void main() {  
  runApp(MyApp());  
}
```

`runApp()` prend un widget en argument et l'affiche sur l'écran.

# ARBRE DE WIDGETS DE BASE

L'arbre de widgets est la structure hiérarchique des widgets dans une application Flutter. Chaque application Flutter est composée d'un arbre de widgets imbriqués. Un widget parent peut avoir un ou plusieurs widgets enfants. Les widgets enfants peuvent eux-mêmes avoir d'autres widgets enfants.

# NOTION DE MATERIALAPP ET CUPERTINOAPP

**MaterialApp** est un widget qui implémente le design Material de Google. Il fournit des thèmes, des routes, et d'autres configurations de base.

```
MaterialApp(  
    title: 'Flutter Demo',  
    theme: ThemeData(  
        primarySwatch: Colors.blue,  
    ),  
    home: MyHomePage(),  
)
```

**CupertinoApp** est un widget qui implémente le design iOS de Cupertino. Il est utilisé pour créer des applications avec une apparence iOS native.

```
CupertinoApp(  
    title: 'Flutter Demo',  
    home: MyHomePage(),  
)
```

# NOTIONS DE WIDGETS EN FLUTTER

# DÉFINITION D'UN WIDGET

En Flutter, un widget est une unité de base de l'interface utilisateur. Chaque élément de l'interface est un widget. Les widgets définissent la structure, le style et les interactions de l'interface. Les widgets peuvent être imbriqués pour créer des interfaces complexes. Il existe deux types principaux de widgets : StatelessWidget et StatefulWidget.

# TYPES DE WIDGETS (STATELESSWIDGET, STATEFULWIDGET)

# STATELESSWIDGET

Un StatelessWidget est un widget immuable. Il ne change pas d'état après sa création. Utilisé pour les éléments statiques de l'interface. Exemple : Texte, Icône.

# STATEFULWIDGET

Un StatefulWidget est un widget mutable. Il peut changer d'état pendant l'exécution de l'application. Utilisé pour les éléments dynamiques de l'interface. Exemple : Bouton, Formulaire.

# CYCLE DE VIE DES WIDGETS

# STATELESSWIDGET

1. `build()`: Méthode appelée pour construire le widget.

# STATEFULWIDGET

1. `createState()`: Crée un objet State.
2. `initState()`: Initialisation de l'état.
3. `build()`: Construit le widget.
4. `setState()`: Met à jour l'état.
5. `dispose()`: Nettoyage des ressources.

# COMPOSITION DES WIDGETS

Les widgets peuvent être composés pour créer des interfaces complexes. Les widgets parents contiennent des widgets enfants. Les widgets enfants héritent des propriétés des widgets parents. Exemple :

```
Column(  
  children: [  
    Text('Hello'),  
    Icon(Icons.star),  
  ],  
)
```

# UTILISATION DES WIDGETS INTÉGRÉS

Flutter fournit de nombreux widgets intégrés. Ils couvrent des besoins courants comme les mises en page, les boutons, les champs de texte. Exemples de widgets intégrés :

- Container
- Row
- Column
- Text
- Button

# CRÉATION DE WIDGETS PERSONNALISÉS

Créer un widget personnalisé en étendant `StatelessWidget` ou  `StatefulWidget`. Définir la méthode `build` pour retourner l'arbre de widgets. Exemple de widget personnalisé :

```
class MyWidget extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return Text('Mon Widget Personnalisé');  
  }  
}
```

# NAVIGATION DE BASE ENTRE LES ÉCRANS

# INTRODUCTION À LA NAVIGATION

La navigation permet de passer d'un écran à un autre. Flutter utilise un système de routes pour gérer la navigation. Chaque écran est défini comme une route. Les routes peuvent être nommées ou anonymes. Navigator est le widget principal pour la navigation.

# CRÉATION DE ROUTES

Une route est généralement un widget. Elle peut être définie comme une fonction renvoyant un widget.  
Exemple de route :

```
Widget routeExample() {  
    return Scaffold(  
        appBar: AppBar(title: Text('Route Example')),  
        body: Center(child: Text('This is a route')),  
    );  
}
```

# DÉFINITION DES ROUTES DANS MATERIALAPP

Les routes sont définies dans MaterialApp. Utilisez la propriété `routes` pour définir les routes nommées.  
Exemple :

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

# NAVIGATION AVEC NAVIGATOR.PUSH

`Navigator.push` permet de naviguer vers une nouvelle route. Utilisation :

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

`Navigator.push` ajoute la nouvelle route en haut de la pile.

# NAVIGATION AVEC NAVIGATOR.POP

Navigator.pop permet de revenir à la route précédente. Utilisation :

```
Navigator.pop(context);
```

Navigator.pop retire la route actuelle de la pile.

# UTILISATION DE NAVIGATOR

# INTRODUCTION À NAVIGATOR

**Navigator** est un widget de Flutter utilisé pour gérer la navigation entre les écrans. Il permet de pousser et de retirer des routes (pages) de la pile de navigation. Chaque application Flutter a un **Navigator** par défaut. Les routes sont définies comme des widgets. Le **Navigator** utilise une pile pour gérer les routes.

# MÉTHODES PUSH ET POP

`Navigator.push` ajoute une nouvelle route à la pile de navigation. Syntaxe :

```
Navigator.push(  
  context,  
  MaterialPageRoute(builder: (context) => NewScreen()),  
)
```

`Navigator.pop` retire la route actuelle de la pile. Syntaxe :

```
Navigator.pop(context);
```

# PASSAGE DE DONNÉES ENTRE LES ÉCRANS

Les données peuvent être passées lors de l'appel à `Navigator.push`. Exemple :

```
Navigator.push(  
  context,  
  MaterialPageRoute(  
    builder: (context) => NewScreen(data: 'Hello'),  
  ),  
);
```

Le nouvel écran peut accéder aux données via ses paramètres.

# GESTION DE LA PILE DE NAVIGATION

La pile de navigation est une structure LIFO (Last In, First Out). Chaque appel à `Navigator.push` ajoute une route au sommet de la pile. Chaque appel à `Navigator.pop` retire la route du sommet de la pile. La gestion de la pile permet de contrôler l'historique de navigation.

# UTILISATION DE NAVIGATOR.PUSHREPLACEMENT

Navigator.pushReplacement remplace la route actuelle par une nouvelle route. Syntaxe :

```
Navigator.pushReplacement(  
  context,  
  MaterialPageRoute(builder: (context) => NewScreen()),  
)
```

Cette méthode ne conserve pas l'ancienne route dans la pile.

# UTILISATION DE NAVIGATOR.POPUNTIL

`Navigator.popUntil` retire les routes jusqu'à une route spécifique. Syntaxe :

```
Navigator.popUntil(context, ModalRoute.withName('/home'));
```

Cette méthode permet de revenir à une route spécifique en supprimant les routes intermédiaires.

# ROUTES NOMMÉES

# DÉFINITION DES ROUTES NOMMÉES

Les routes nommées permettent de définir des chemins de navigation. Elles simplifient la gestion de la navigation dans une application Flutter. Chaque route nommée est associée à une chaîne de caractères unique. Les routes nommées sont définies dans le widget `MaterialApp`. Elles facilitent la navigation entre les différentes pages de l'application.

# CONFIGURATION DES ROUTES NOMMÉES DANS MATERIALAPP

Pour configurer les routes nommées, utilisez la propriété `routes` de `MaterialApp`. Définissez un `Map` où chaque clé est une chaîne de caractères représentant le nom de la route. Les valeurs sont des fonctions qui retournent des widgets. Exemple :

```
MaterialApp(  
  routes: {  
    '/': (context) => HomePage(),  
    '/second': (context) => SecondPage(),  
  },  
);
```

# NAVIGATION AVEC DES ROUTES NOMMÉES

Pour naviguer avec des routes nommées, utilisez la méthode `Navigator.pushNamed`. Elle prend en paramètre le contexte et le nom de la route. Exemple :

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

# PASSER DES ARGUMENTS AVEC DES ROUTES NOMMÉES

Pour passer des arguments, utilisez la méthode `Navigator.pushNamed`. Ajoutez un argument `arguments` à la méthode. Exemple :

```
Navigator.pushNamed(  
  context,  
  '/second',  
  arguments: 'Hello, World!',  
);
```

# GESTION DES ERREURS DE NAVIGATION AVEC DES ROUTES NOMMÉES

Pour gérer les erreurs de navigation, utilisez la propriété `onUnknownRoute` de `MaterialApp`. Elle permet de définir une route de secours en cas de route inconnue. Exemple :

```
MaterialApp(  
    onUnknownRoute: (settings) {  
        return MaterialPageRoute(builder: (context) => ErrorPage());  
    },  
);
```

# GESTION DES ROUTES DYNAMIQUES

# DÉFINITION DES ROUTES DYNAMIQUES

Les routes dynamiques permettent de naviguer vers des écrans dont la destination peut changer. Elles sont définies au moment de l'exécution, contrairement aux routes nommées. Elles permettent de passer des paramètres dynamiques à l'écran cible. Utile pour des interfaces utilisateur complexes et flexibles. Permet de gérer la navigation en fonction des états de l'application.

# CONFIGURATION DES ROUTES DYNAMIQUES

Les routes dynamiques sont configurées dans le widget `MaterialApp`. Utilisez la propriété `onGenerateRoute` pour définir les routes dynamiques. Cette propriété prend une fonction qui retourne un `Route`. La fonction `onGenerateRoute` est appelée pour chaque navigation. Elle permet de contrôler la création des routes de manière dynamique.

# UTILISATION DE MATERIALPAGEROUTE

**MaterialPageRoute** est utilisé pour créer des routes dynamiques. Il permet de spécifier l'écran cible et les paramètres. Exemple de syntaxe :

```
MaterialPageRoute(  
    builder: (context) => TargetScreen(param: value),  
);
```

# UTILISATION DE NAVIGATOR.PUSH POUR LES ROUTES DYNAMIQUES

`Navigator.push` permet de naviguer vers une route dynamique. Exemple de syntaxe :

```
Navigator.push(  
  context,  
  MaterialPageRoute(builder: (context) => TargetScreen(param: value)),  
)
```

Utilisez `Navigator.push` pour ajouter la nouvelle route à la pile de navigation.

# GESTION DES PARAMÈTRES DANS LES ROUTES DYNAMIQUES

Les paramètres peuvent être passés via le constructeur de l'écran cible. Exemple :

```
class TargetScreen extends StatelessWidget {  
    final String param;  
    TargetScreen({required this.param});  
    ...  
}
```

Lors de la navigation, passez les paramètres nécessaires.

# GESTION DES RETOURS DE DONNÉES AVEC LES ROUTES DYNAMIQUES

Utilisez `Navigator.pop` pour retourner des données à l'écran précédent. Exemple de syntaxe :

```
Navigator.pop(context, returnValue);
```

Pour recevoir les données, utilisez `Navigator.push` avec `then` :

```
Navigator.push(...).then((returnValue) {  
  // Utilisez returnValue ici  
});
```

# PASSER DES DONNÉES ENTRE LES ÉCRANS

# UTILISATION DE NAVIGATOR

Navigator est utilisé pour gérer la navigation entre les écrans. Pour naviguer vers un nouvel écran, utilisez `Navigator.push`.

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

Pour revenir à l'écran précédent, utilisez `Navigator.pop`.

```
Navigator.pop(context);
```

# PASSAGE DE PARAMÈTRES VIA NAVIGATOR.PUSH

Vous pouvez passer des paramètres en utilisant le constructeur de l'écran.

```
Navigator.push(  
  context,  
  MaterialPageRoute(  
    builder: (context) => SecondScreen(data: 'Hello'),  
  ),  
);
```

# UTILISATION DE ROUTESETTINGS

RouteSettings permet de passer des arguments lors de la navigation.

```
Navigator.push(  
  context,  
  MaterialPageRoute(  
    builder: (context) => SecondScreen(),  
    settings: RouteSettings(arguments: 'Hello'),  
  ),  
);
```

Pour récupérer les arguments, utilisez `ModalRoute.of`.

```
final args = ModalRoute.of(context)!.settings.arguments as String;
```

# UTILISATION DE CONSTRUCTORS POUR PASSER DES DONNÉES

Les constructors permettent de passer des données directement à l'initialisation de l'écran.

```
class SecondScreen extends StatelessWidget {  
    final String data;  
  
    SecondScreen({required this.data});  
  
    @override  
    Widget build(BuildContext context) {  
        return Scaffold(  
            body: Center(child: Text(data)),  
        );  
    }  
}
```

# GESTION DES ARGUMENTS AVEC MODALROUTE.OF

ModalRoute.of permet d'accéder aux arguments passés via RouteSettings.

```
@override
Widget build(BuildContext context) {
  final args = ModalRoute.of(context)!.settings.arguments as String;

  return Scaffold(
    body: Center(child: Text(args)),
  );
}
```

# UTILISATION DE NAMED ROUTES POUR PASSER DES DONNÉES

Named Routes permettent de gérer la navigation avec des noms de routes.

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

Pour récupérer les arguments dans une Named Route :

```
@override  
Widget build(BuildContext context) {  
  final args = ModalRoute.of(context)!.settings.arguments as String;  
  
  return Scaffold(  
    body: Center(child: Text(args)),  
  );  
}
```

# RETOURNER DES DONNÉES AUX ÉCRANS PRÉCÉDENTS

# UTILISATION DE NAVIGATOR.POP

Pour retourner des données à l'écran précédent, utilisez `Navigator.pop()`.

```
Navigator.pop(context, result);
```

`context` est le contexte de l'écran actuel.

`result` est la donnée que vous souhaitez retourner.

# PASSAGE DE DONNÉES AVEC NAVIGATOR.POP

Exemple de passage de données avec `Navigator.pop`:

```
Navigator.pop(context, 'Hello from the second screen!');
```

Le deuxième argument de `Navigator.pop` contient les données retournées.

# GESTION DES RÉSULTATS RETOURNÉS

Pour gérer les résultats retournés, utilisez `await` avec `Navigator.push`.

```
final result = await Navigator.push(  
  context,  
  MaterialPageRoute(builder: (context) => SecondScreen()),  
);
```

Le résultat retourné par `Navigator.pop` sera stocké dans `result`.

# UTILISATION DE FUTURE POUR LES DONNÉES RETOURNÉES

Navigator.push retourne un Future qui contient les données renvoyées.

```
Future<String> navigateAndReturnData(BuildContext context) async {
  final result = await Navigator.push(
    context,
    MaterialPageRoute(builder: (context) => SecondScreen()),
  );
  return result;
}
```

Utilisez await pour attendre le résultat renvoyé.

# UTILISATION DE MATERIALPAGEROUTE

# NOTION DE MATERIALPAGEROUTE

**MaterialPageRoute** est une classe de Flutter utilisée pour gérer la navigation. Elle permet de définir une route basée sur le design Material. Elle est souvent utilisée pour naviguer entre les écrans. Elle permet de spécifier la transition d'une page à une autre. Elle est compatible avec les animations de transition par défaut de Flutter. Utilisée couramment dans les applications Flutter pour une navigation fluide.

# CRÉATION DE MATERIALPAGEROUTE

Pour créer une `MaterialPageRoute`, utilisez le constructeur suivant :

```
MaterialPageRoute(  
  builder: (context) => SecondPage(),  
) ;
```

Vous pouvez ensuite utiliser `Navigator` pour pousser la route :

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

# PASSAGE DE PARAMÈTRES AVEC MATERIALPAGEROUTE

Vous pouvez passer des paramètres à la nouvelle page via le constructeur :

```
Navigator.push(  
  context,  
  MaterialPageRoute(  
    builder: (context) => SecondPage(data: 'Hello'),  
  ),  
);
```

Dans **SecondPage**, récupérez les paramètres dans le constructeur :

```
class SecondPage extends StatelessWidget {  
  final String data;  
  
  SecondPage({required this.data});  
}
```

# GESTION DES TRANSITIONS AVEC MATERIALPAGEROUTE

`MaterialPageRoute` utilise des transitions par défaut basées sur le design Material. Vous pouvez personnaliser les transitions en utilisant le paramètre `settings`. Pour des transitions avancées, utilisez `PageRouteBuilder`.

Exemple de transition personnalisée :

```
Navigator.push(  
  context,  
  MaterialPageRoute(  
    builder: (context) => SecondPage(),  
    settings: PageRouteSettings(  
      pageBuilder: (context, animation, secondaryAnimation) => SecondPage(),  
      transitionsBuilder: (context, animation, secondaryAnimation, child) {  
        return FadeTransition(opacity: animation, child: child);  
      },  
    ),  
  );
```

# UTILISATION DE NAVIGATOR AVEC MATERIALPAGEROUTE

Navigator est utilisé pour gérer la pile de routes. Pour naviguer vers une nouvelle page, utilisez Navigator.push :

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

Pour revenir à la page précédente, utilisez Navigator.pop :

```
Navigator.pop(context);
```

Vous pouvez également utiliser Navigator.pushReplacement pour remplacer la route actuelle.

# UTILISATION DE CUPERTINOPAGEROUTE

# INTRODUCTION À CUPERTINOPAGEROUTE

**CupertinoPageRoute** est une classe de Flutter. Elle permet de créer des transitions de style iOS. Utilisée pour une navigation cohérente avec les designs iOS. Fait partie du package **cupertino** de Flutter. Elle offre des animations de transition spécifiques à iOS. Facile à intégrer dans une application Flutter existante.

# DIFFÉRENCES ENTRE CUPERTINOPAGEROUTE ET MATERIALPAGEROUTE

Caractéristique	CupertinoPageRoute	MaterialPageRoute
Style de transition	iOS	Android
Animation de transition	Glissement de droite à gauche	Glissement de bas en haut
Compatibilité	iOS	Android
Personnalisation	Limité	Plus flexible
Thème par défaut	Cupertino (iOS)	Material (Android)

# CRÉATION D'UNE ROUTE AVEC CUPERTINOPAGEROUTE

Pour créer une route avec `CupertinoPageRoute` :

```
Navigator.push(  
  context,  
  CupertinoPageRoute(builder: (context) => NewScreen()),  
)
```

Utilisez `CupertinoPageRoute` pour des transitions iOS. Remplace `MaterialPageRoute` par `CupertinoPageRoute`.

# GESTION DE LA NAVIGATION AVEC CUPERTINOPAGEROUTE

Pour naviguer entre les écrans :

```
Navigator.push(  
  context,  
  CupertinoPageRoute(builder: (context) => NewScreen()),  
) ;
```

Pour revenir à l'écran précédent :

```
Navigator.pop(context);
```

Utilisez **Navigator** pour gérer la pile de navigation. **CupertinoPageRoute** assure des transitions iOS.

# PERSONNALISATION DES ANIMATIONS DE TRANSITION AVEC CUPERTINOPAGEROUTE

CupertinoPageRoute offre des animations par défaut. Pour personnaliser, utilisez CupertinoPageRoute avec des paramètres. Exemple de personnalisation :

```
Navigator.push(  
  context,  
  CupertinoPageRoute(  
    builder: (context) => NewScreen(),  
    fullscreenDialog: true,  
  ),  
);
```

fullscreenDialog affiche l'écran en plein écran. Personnalisez les animations pour une meilleure expérience utilisateur.

# ANIMATION DE TRANSITION ENTRE LES ÉCRANS

# INTRODUCTION AUX ANIMATIONS DE TRANSITION

Les animations de transition améliorent l'expérience utilisateur. Elles rendent les changements d'écran plus fluides et attrayants. Flutter propose plusieurs méthodes pour créer ces animations. Les animations peuvent être simples ou complexes selon les besoins. Les animations de transition sont souvent utilisées pour les changements d'écran. Elles peuvent inclure des effets de glissement, de fondu, de rotation, etc. Les animations peuvent être personnalisées pour répondre aux besoins spécifiques. Flutter fournit des widgets et des classes dédiés pour les animations. Les principales méthodes incluent PageRouteBuilder, Hero Animation, et AnimatedSwitcher. Comprendre ces méthodes est essentiel pour créer des applications Flutter interactives.

# UTILISATION DE PAGEROUTEBUILDER

PageRouteBuilder permet de créer des transitions personnalisées. Il utilise une fonction de transition pour animer les changements d'écran. La fonction de transition prend en paramètre Animation et Widget.

Exemple de PageRouteBuilder :

```
Navigator.push(context, MaterialPageRoute(  
    pageBuilder: (context, animation, secondaryAnimation) => SecondPage(),  
    transitionsBuilder: (context, animation, secondaryAnimation, child) {  
        return FadeTransition(opacity: animation, child: child);  
    },  
));
```

# ANIMATION DE TRANSITION PERSONNALISÉE

Les animations de transition personnalisées offrent plus de flexibilité. On peut combiner plusieurs animations pour créer des effets uniques. Exemple de transition personnalisée avec PageRouteBuilder :

```
Navigator.push(context, PageRouteBuilder(  
    pageBuilder: (context, animation, secondaryAnimation) => SecondPage(),  
    transitionsBuilder: (context, animation, secondaryAnimation, child) {  
        var begin = Offset(0.0, 1.0);  
        var end = Offset.zero;  
        var curve = Curves.ease;  
  
        var tween = Tween(begin: begin, end: end).chain(CurveTween(curve: curve));  
  
        return SlideTransition(position: animation.drive(tween), child: child);  
    },  
);
```

# UTILISATION DE HERO ANIMATION

Hero Animation permet de créer des animations partagées entre deux écrans. Il utilise le widget Hero pour animer les transitions. Le widget Hero doit avoir le même tag sur les deux écrans. Exemple d'utilisation de Hero Animation :

```
Hero(  
  tag: 'hero-tag',  
  child: Image.asset('assets/image.png'),  
)
```

# UTILISATION DE ANIMATEDSWITCHER

AnimatedSwitcher permet de changer de widget avec une animation. Il utilise une transition animée pour remplacer le widget enfant. Exemple d'utilisation de AnimatedSwitcher :

```
AnimatedSwitcher(  
  duration: const Duration(seconds: 1),  
  child: _myWidget,  
);
```

# ANIMATION D'ENTRÉE ET DE SORTIE

Les animations d'entrée et de sortie améliorent les transitions de widget. On peut utiliser des widgets comme FadeTransition et SlideTransition. Exemple d'animation d'entrée et de sortie :

```
FadeTransition(  
    opacity: _animation,  
    child: MyWidget(),  
);
```

# Contrôle de la durée et de la courbe de l'animation

La durée et la courbe de l'animation contrôlent la vitesse et le style. On peut utiliser Duration et Curve pour personnaliser ces aspects. Exemple de contrôle de la durée et de la courbe :

```
var animation = CurvedAnimation(  
  parent: controller,  
  curve: Curves.easeInOut,  
)  
  
controller.duration = Duration(seconds: 2);
```

# GESTION DES PRÉFÉRENCES UTILISATEUR AVEC SHARED PREFERENCES

# INTRODUCTION À SHARED PREFERENCES

SharedPreferences est une bibliothèque Flutter pour stocker des données simples. Elle permet de sauvegarder des préférences utilisateur localement. Les données sont stockées sous forme de paires clé-valeur. Utilisée pour stocker des informations comme les paramètres de l'application. Elle est idéale pour des données simples et légères. Non adaptée pour des données complexes ou volumineuses.

# INSTALLATION DE SHARED PREFERENCES

Pour installer SharedPreferences, ajoutez la dépendance dans `pubspec.yaml` :

```
dependencies:  
  shared_preferences: ^2.0.6
```

Ensuite, exédez `flutter pub get` pour télécharger la bibliothèque.

# SAUVEGARDE DE DONNÉES SIMPLES

Pour sauvegarder des données simples, utilisez les méthodes `set` :

```
final prefs = await SharedPreferences.getInstance();
prefs.setString('username', 'JohnDoe');
prefs.setInt('age', 25);
prefs.setBool('isLoggedIn', true);
```

# LECTURE DE DONNÉES SAUVEGARDÉES

Pour lire des données sauvegardées, utilisez les méthodes `get` :

```
final prefs = await SharedPreferences.getInstance();
String? username = prefs.getString('username');
int? age = prefs.getInt('age');
bool? isLoggedIn = prefs.getBool('isLoggedIn');
```

# SUPPRESSION DE DONNÉES

Pour supprimer des données, utilisez la méthode `remove`:

```
final prefs = await SharedPreferences.getInstance();
prefs.remove('username');
```

Vous pouvez également utiliser `clear` pour tout supprimer:

```
prefs.clear();
```

# GESTION DES TYPES DE DONNÉES (STRING, INT, BOOL, ETC.)

SharedPreferences supporte différents types de données :

- String : `prefs.setString('key', 'value')`
- int : `prefs.setInt('key', value)`
- bool : `prefs.setBool('key', value)`
- double : `prefs.setDouble('key', value)`
- List : `prefs.setStringList('key', value)`

# UTILISATION DE SHARED PREFERENCES DANS DES WIDGETS

Utilisez SharedPreferences dans des widgets pour sauvegarder et lire des préférences :

```
class MyWidget extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return FutureBuilder(  
      future: SharedPreferences.getInstance(),  
      builder: (context, snapshot) {  
        if (snapshot.connectionState == ConnectionState.done) {  
          final prefs = snapshot.data as SharedPreferences;  
          String? username = prefs.getString('username');  
          return Text(username ?? 'No username saved');  
        }  
        return CircularProgressIndicator();  
      },  
    );  
  }  
}
```

# BONNES PRATIQUES ET LIMITATIONS DE SHARED PREFERENCES

## BONNES PRATIQUES

- Utilisez des clés descriptives pour les préférences.
- Sauvegardez uniquement des données simples et légères.
- Utilisez des méthodes asynchrones pour éviter les blocages.

# LIMITATIONS

- Non adapté pour des données volumineuses ou complexes.
- Les données sont stockées en clair, pas de chiffrement par défaut.
- Pas de support pour des types de données complexes.

# STOCKAGE DES PRÉFÉRENCES UTILISATEUR LOCALEMENT

# SAUVEGARDE DES PRÉFÉRENCES UTILISATEUR

Pour sauvegarder les préférences utilisateur localement, Flutter utilise le package `shared_preferences`.

Ajoutez le package à votre fichier `pubspec.yaml` :

```
dependencies:  
  shared_preferences: ^2.0.6
```

Utilisez la méthode `setString`, `setInt`, `setBool`, etc., pour sauvegarder les préférences.

# TYPES DE DONNÉES SUPPORTÉS

`shared_preferences` supporte les types de données suivants :

- `String`
- `int`
- `double`
- `bool`
- `List<String>`

Utilisez les méthodes appropriées pour chaque type de données.

# GESTION DES CLÉS ET VALEURS

Les préférences utilisateur sont stockées sous forme de paires clé-valeur.

Exemple pour sauvegarder un nom d'utilisateur :

```
SharedPreferences prefs = await SharedPreferences.getInstance();  
prefs.setString('username', 'JohnDoe');
```

# MISE À JOUR DES PRÉFÉRENCES

Pour mettre à jour une préférence existante, utilisez la même méthode que pour la sauvegarde.

Exemple pour mettre à jour le nom d'utilisateur :

```
prefs.setString('username', 'JaneDoe');
```

# SUPPRESSION DES PRÉFÉRENCES

Pour supprimer une préférence, utilisez la méthode `remove` :

```
prefs.remove('username');
```

Pour supprimer toutes les préférences, utilisez la méthode `clear` :

```
prefs.clear();
```

# BONNES PRATIQUES DE STOCKAGE LOCAL

- Utilisez des clés descriptives pour éviter les conflits.
- Sauvegardez uniquement les données nécessaires.
- Validez les données avant de les sauvegarder.
- Utilisez des valeurs par défaut lorsque c'est possible.
- Testez les préférences sur différents appareils et versions d'OS.

# LECTURE DES PRÉFÉRENCES UTILISATEUR

# ACCÉDER AUX PRÉFÉRENCES UTILISATEUR

Pour accéder aux préférences utilisateur en Flutter, utilisez le package `shared_preferences`.

Ajoutez la dépendance dans le fichier `pubspec.yaml` :

```
dependencies:  
  shared_preferences: ^2.0.6
```

Importez le package dans votre fichier Dart :

```
import 'package:shared_preferences/shared_preferences.dart';
```

# LIRE DES VALEURS SIMPLES (BOOLÉENS, CHAÎNES, NOMBRES)

Pour lire des valeurs simples, utilisez les méthodes appropriées de `SharedPreferences` :

```
SharedPreferences prefs = await SharedPreferences.getInstance();
bool boolValue = prefs.getBool('key') ?? false;
String stringValue = prefs.getString('key') ?? '';
int intValue = prefs.getInt('key') ?? 0;
double doubleValue = prefs.getDouble('key') ?? 0.0;
```

# LIRE DES LISTES ET DES ENSEMBLES

Pour lire des listes et des ensembles, utilisez les méthodes appropriées de `SharedPreferences` :

```
SharedPreferences prefs = await SharedPreferences.getInstance();
List<String> stringList = prefs.getStringList('key') ?? [];
```

# GÉRER LES VALEURS PAR DÉFAUT

Pour gérer les valeurs par défaut, utilisez l'opérateur de coalescence nulle (??) :

```
SharedPreferences prefs = await SharedPreferences.getInstance();
bool boolValue = prefs.getBool('key') ?? true; // valeur par défaut true
String stringValue = prefs.getString('key') ?? 'default'; // valeur par défaut 'default'
int intValue = prefs.getInt('key') ?? 1; // valeur par défaut 1
double doubleValue = prefs.getDouble('key') ?? 1.0; // valeur par défaut 1.0
```

# DÉTECTION DES CHANGEMENTS DE PRÉFÉRENCES

Pour détecter les changements de préférences, utilisez un Stream ou un ValueNotifier :

```
class PreferencesNotifier extends ValueNotifier<Map<String, dynamic>> {
  PreferencesNotifier() : super({});

  void updatePreferences() async {
    SharedPreferences prefs = await SharedPreferences.getInstance();
    value = {
      'boolValue': prefs.getBool('key') ?? false,
      'stringValue': prefs.getString('key') ?? '',
      'intValue': prefs.getInt('key') ?? 0,
      'doubleValue': prefs.getDouble('key') ?? 0.0,
    };
  }
}
```

# MISE À JOUR DES PRÉFÉRENCES UTILISATEUR

# MÉTHODES DE MISE À JOUR

Pour mettre à jour les préférences utilisateur en Flutter, utilisez le package `shared_preferences`.

```
SharedPreferences prefs = await SharedPreferences.getInstance();  
prefs.setString('username', 'nouveau_nom_utilisateur');
```

# VALIDATION DES DONNÉES

Avant de mettre à jour les préférences, validez les données :

- Vérifiez les types de données.
- Assurez-vous que les valeurs ne sont pas nulles.
- Utilisez des expressions régulières pour valider les formats.

# GESTION DES ERREURS

Pour gérer les erreurs lors de la mise à jour des préférences :

- Utilisez des blocs `try-catch`.
- Affichez des messages d'erreur à l'utilisateur.
- Loggez les erreurs pour le débogage.

# SYNCHRONISATION AVEC LE BACKEND

Pour synchroniser les préférences avec le backend :

- Envoyez une requête HTTP POST/PUT.
- Mettez à jour les préférences locales après une réponse réussie.
- Gérez les conflits en utilisant des horodatages.

# MISE À JOUR ASYNCHRONE

Les mises à jour des préférences sont asynchrones :

```
await prefs.setString('username', 'nouveau_nom_utilisateur');
```

- Utilisez `await` pour attendre la complétion.
- Gérez les futures avec `then` et `catchError`.

# NOTIFICATION DES CHANGEMENTS

Pour notifier les changements de préférences :

- Utilisez des `ValueNotifier` ou `ChangeNotifier`.
- Abonnez-vous aux changements dans votre UI.
- Appelez `notifyListeners()` après la mise à jour.

# SUPPRESSION DES PRÉFÉRENCES UTILISATEUR

# MÉTHODES DE SUPPRESSION

Flutter offre plusieurs méthodes pour supprimer des préférences utilisateur :

- `remove(key)`: Supprime une préférence spécifique.
- `clear()`: Supprime toutes les préférences.

Exemple :

```
prefs.remove('username');  
prefs.clear();
```

# GESTION DES ERREURS LORS DE LA SUPPRESSION

Lors de la suppression des préférences, il est crucial de gérer les erreurs :

- Utiliser des blocs **try - catch** pour capturer les exceptions.
- Afficher des messages d'erreur à l'utilisateur.

Exemple :

```
try {
    prefs.remove('username');
} catch (e) {
    print('Erreur lors de la suppression : $e');
}
```

# VÉRIFICATION DE L'EXISTENCE DES PRÉFÉRENCES AVANT SUPPRESSION

Avant de supprimer une préférence, vérifiez son existence :

- Utiliser `containsKey(key)` pour vérifier.

Exemple :

```
if (prefs.containsKey('username')) {  
    prefs.remove('username');  
}
```

# IMPACT DE LA SUPPRESSION SUR L'APPLICATION

La suppression des préférences peut avoir des impacts :

- Perte de données utilisateur.
- Nécessité de reconfigurer l'application.
- Modification de l'expérience utilisateur.

# SUPPRESSION CONDITIONNELLE DES PRÉFÉRENCES

La suppression peut être conditionnelle :

- Basée sur des critères spécifiques.
- Utiliser des structures de contrôle comme `if-else`.

Exemple :

```
if (prefs.getInt('loginCount') > 5) {  
    prefs.remove('username');  
}
```

# UTILISATION DE BIBLIOTHÈQUES TIERCES POUR LA SUPPRESSION

Pour des fonctionnalités avancées, utilisez des bibliothèques tierces :

- `shared_preferences` pour des opérations de base.
- `hive` pour des besoins plus complexes.

Exemple avec `shared_preferences` :

```
import 'package:shared_preferences/shared_preferences.dart';

final prefs = await SharedPreferences.getInstance();
prefs.remove('username');
```

# SAUVEGARDE DES ÉTATS DE L'APPLICATION

# NOTION D'ÉTAT

L'état d'une application représente les données dynamiques qui peuvent changer au cours de l'exécution. En Flutter, l'état est souvent géré par des widgets. Il existe deux types de widgets : StatefulWidget et StatelessWidget. StatefulWidget peut changer d'état au cours du temps. StatelessWidget ne peut pas changer d'état.

# STATEFULWIDGET VS STATELESSWIDGET

## **StatefulWidget :**

- Peut changer d'état.
- Utilise un objet **State** pour gérer l'état.

## **StatelessWidget :**

- Ne peut pas changer d'état.
- Utilisé pour des contenus statiques.

# MÉTHODES DE SAUVEGARDE D'ÉTAT

Il existe plusieurs méthodes pour sauvegarder l'état d'une application Flutter :

- Shared Preferences
- Base de données locale comme SQLite
- Sauvegarde dans le cloud

# SHARED PREFERENCES

**SharedPreferences** est utilisé pour stocker des paires clé-valeur simples. Il est idéal pour stocker des préférences utilisateur et des données légères. Exemple d'utilisation :

```
SharedPreferences prefs = await SharedPreferences.getInstance();  
prefs.setInt('counter', 10);
```

# SQLITE

SQLite est une base de données relationnelle intégrée. Elle est idéale pour stocker des données structurées. Flutter utilise le package **sqflite** pour interagir avec SQLite. Exemple d'utilisation :

```
Database db = await openDatabase('my_db.db');
await db.insert('table', {'column': 'value'});
```

# SAUVEGARDE DANS LE CLOUD

Sauvegarder l'état dans le cloud permet la persistance des données sur plusieurs appareils. Utilisation courante de services comme Firebase Firestore. Exemple d'utilisation :

```
Firestore.instance.collection('users').add({'name': 'John Doe'});
```

# RESTAURATION DE L'ÉTAT

La restauration de l'état consiste à recharger les données sauvegardées. Utiliser les mêmes méthodes que pour la sauvegarde. Exemple avec **SharedPreferences** :

```
SharedPreferences prefs = await SharedPreferences.getInstance();  
int counter = prefs.getInt('counter') ?? 0;
```

# GESTION DES ÉTATS COMPLEXES

Pour les états complexes, utiliser des solutions comme :

- Provider
- Bloc
- Redux Ces solutions facilitent la gestion et la mise à jour de l'état.

# BONNES PRATIQUES POUR LA SAUVEGARDE DES ÉTATS

- Sauvegarder régulièrement l'état critique.
- Utiliser des méthodes adaptées à la taille et à la complexité des données.
- Tester la restauration de l'état pour assurer la fiabilité.
- Sécuriser les données sensibles lors de la sauvegarde.

# MEILLEURES PRATIQUES POUR LE ROUTAGE EN FLUTTER

# UTILISATION DE LA BIBLIOTHÈQUE NAVIGATOR

La bibliothèque **Navigator** permet de gérer la navigation entre les écrans. Utilisation de **push** pour ajouter une nouvelle route à la pile de navigation. Utilisation de **pop** pour retirer la route actuelle de la pile. Exemple de navigation :

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

# DÉFINITION DES ROUTES NOMMÉES

Les routes nommées permettent de définir des chemins de navigation plus clairs. Définition des routes dans `MaterialApp` :

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

# UTILISATION DE LA BIBLIOTHÈQUE AUTO\_ROUTE

`auto_route` facilite la gestion des routes complexes. Définition des routes dans un fichier séparé :

```
@MaterialAutoRouter(  
  routes: <AutoRoute>[  
    MaterialRoute(page: HomeScreen, initial: true),  
    MaterialRoute(page: SecondScreen),  
  ],  
)  
class $AppRouter {}
```

# GESTION DES TRANSITIONS D'ÉCRAN

Personnalisation des transitions d'écran pour une meilleure UX. Utilisation de PageRouteBuilder :

```
Navigator.push(  
  context,  
  PageRouteBuilder(  
    pageBuilder: (context, animation, secondaryAnimation) => SecondScreen(),  
    transitionsBuilder: (context, animation, secondaryAnimation, child) {  
      return FadeTransition(opacity: animation, child: child);  
    },  
  ),  
);
```

# UTILISATION DES ROUTES GÉNÉRÉES DYNAMIQUEMENT

Création de routes en fonction des conditions runtime. Utilisation de `onGenerateRoute` :

```
MaterialApp(  
    onGenerateRoute: (settings) {  
        if (settings.name == '/second') {  
            return MaterialPageRoute(builder: (context) => SecondScreen());  
        }  
        return null;  
    },  
);
```

# SÉPARATION DES PRÉOCCUPATIONS (ROUTES ET LOGIQUE MÉTIER)

Séparation des routes et de la logique métier pour une meilleure maintenabilité. Utilisation de fichiers séparés pour les routes et la logique métier. Exemple :

```
// routes.dart
final routes = {
  '/': (context) => HomeScreen(),
  '/second': (context) => SecondScreen(),
};

// main.dart
MaterialApp(
  routes: routes,
);
```

# UTILISATION DES PACKAGES DE GESTION DE L'ÉTAT POUR LE ROUTAGE

Intégration des packages de gestion de l'état pour une navigation plus fluide. Exemples de packages : **provider**, **bloc**, **riverpod**. Exemple avec **provider** :

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

# CONFIGURATION DES ROUTES INITIALES ET DE FALBACK

Définition des routes initiales et de fallback pour gérer les erreurs de navigation. Configuration dans `MaterialApp` :

```
MaterialApp(  
    initialRoute: '/',  
    onUnknownRoute: (settings) {  
        return MaterialPageRoute(builder: (context) => UnknownScreen());  
    },  
);
```

# DÉBOGAGE DES PROBLÈMES DE ROUTAGE

# IDENTIFICATION DES PROBLÈMES COURANTS DE ROUTAGE

- Routes mal définies
- Problèmes de navigation conditionnelle
- Routes non trouvées (404)
- Boucles de navigation infinies
- Conflits de noms de routes
- Erreurs d'arguments de route
- Problèmes de performance lors de la navigation
- Transitions d'écran incorrectes

# UTILISATION DE LA CONSOLE POUR LE DÉBOGAGE

- Utiliser `print()` pour afficher les routes
- Vérifier les erreurs dans la console
- Utiliser `debugPrint()` pour des messages détaillés
- Analyser les logs d'erreur
- Vérifier les messages d'avertissement
- Utiliser les commandes de débogage de Flutter
- Analyser les exceptions levées

# OUTILS DE DÉBOGAGE INTÉGRÉS DE FLUTTER

- Flutter DevTools
- Widget Inspector
- Timeline pour analyser les performances
- Profiler pour analyser les ressources
- Memory pour analyser l'utilisation de la mémoire
- Network pour analyser les requêtes réseau
- Console pour afficher les logs

# JOURNALISATION DES ROUTES

- Utiliser `NavigatorObserver` pour suivre les routes
- Implémenter un `RouteObserver`
- Enregistrer les transitions de route
- Utiliser des outils de journalisation comme `logger`
- Analyser les logs pour détecter les anomalies
- Configurer des niveaux de journalisation
- Stocker les logs pour une analyse ultérieure

# RÉSOLUTION DES ERREURS DE NAVIGATION

- Vérifier les noms des routes
- Assurer la correspondance des arguments de route
- Utiliser des routes nommées correctement
- Gérer les routes inconnues avec `onUnknownRoute`
- Tester la navigation avec des cas d'utilisation réels
- Vérifier les conditions de navigation
- Utiliser des assertions pour vérifier les routes

# DÉBOGAGE DES TRANSITIONS D'ÉCRAN

- Utiliser des animations de transition correctement
- Vérifier les durées et courbes d'animation
- Utiliser `Hero` pour les transitions complexes
- Déboguer les animations avec Flutter DevTools
- Vérifier les états des widgets pendant les transitions
- Utiliser `TransitionBuilder` pour personnaliser les transitions
- Tester les transitions sur différents appareils

# GESTION DES EXCEPTIONS LIÉES AU ROUTAGE

- Utiliser `try-catch` pour capturer les exceptions
- Analyser les messages d'erreur
- Utiliser des outils comme `Sentry` pour capturer les exceptions
- Gérer les erreurs de navigation avec des écrans d'erreur
- Vérifier les conditions préalables à la navigation
- Utiliser des assertions pour éviter les erreurs
- Documenter et corriger les erreurs fréquentes

# TEST DU ROUTAGE ET DES PRÉFÉRENCES UTILISATEUR

# INTRODUCTION AUX TESTS EN FLUTTER

- Les tests en Flutter permettent de vérifier le bon fonctionnement de l'application.
- Ils aident à identifier et corriger les bugs avant la mise en production.
- Types de tests : tests unitaires, tests widget, tests d'intégration.
- Les tests unitaires vérifient les fonctionnalités individuelles.
- Les tests widget vérifient l'interface utilisateur.
- Les tests d'intégration vérifient le comportement de l'application dans son ensemble.
- Flutter offre des outils intégrés pour faciliter les tests.

# OUTILS DE TEST EN FLUTTER

- `flutter_test`: Package de base pour les tests unitaires et widget.
- `flutter_driver`: Pour les tests d'intégration.
- `mockito`: Pour le mocking des dépendances.
- `shared_preferences`: Pour tester les préférences utilisateur.
- `integration_test`: Pour tests d'intégration sur appareils réels.
- `test`: Package général pour les tests Dart.

# TEST DES ROUTES AVEC WIDGETTESTER

- `WidgetTester` permet de tester les widgets et les routes.
- Utilisation de `pumpWidget` pour charger l'application.
- Navigation avec `tester.tap` et `tester.pumpAndSettle`.
- Vérification des routes avec `expect` et `find`.

```
await tester.tap(find.byIcon(Icons.navigate_next));  
await tester.pumpAndSettle();  
expect(find.text('Next Page'), findsOneWidget);
```

# TEST DES PRÉFÉRENCES UTILISATEUR AVEC SHARED PREFERENCES

- SharedPreferences permet de stocker les préférences utilisateur.
- Utilisation de `setString`, `getInt`, `setBool`, etc.
- Mocking de SharedPreferences pour les tests.

```
SharedPreferences.setMockInitialValues({});  
final prefs = await SharedPreferences.getInstance();  
await prefs.setString('key', 'value');  
expect(prefs.getString('key'), 'value');
```

# ÉCRITURE DE TESTS UNITAIRES POUR LE ROUTAGE

- Tests unitaires pour vérifier la logique de routage.
- Utilisation de `NavigatorObserver` pour suivre la navigation.
- Vérification des méthodes `push`, `pop`, etc.

```
final observer = MockNavigatorObserver();
await tester.pumpWidget(MyApp(observer: observer));
verify(observer.didPush(any, any)).called(1);
```

# ÉCRITURE DE TESTS UNITAIRES POUR LES PRÉFÉRENCES UTILISATEUR

- Tests unitaires pour vérifier la gestion des préférences.
- Mocking de `SharedPreferences` pour isoler les tests.
- Vérification des méthodes `setString`, `getInt`, etc.

```
SharedPreferences.setMockInitialValues({'key': 'value'});
final prefs = await SharedPreferences.getInstance();
expect(prefs.getString('key'), 'value');
```

# UTILISATION DE MOCKING POUR LES TESTS DE ROUTAGE

- Mocking des dépendances pour isoler les tests.
- Utilisation de `mockito` pour créer des objets mock.
- Vérification des interactions avec les objets mock.

```
class MockNavigatorObserver extends Mock implements NavigatorObserver {}  
final observer = MockNavigatorObserver();  
await tester.pumpWidget(MyApp(observer: observer));  
verify(observer.didPush(any, any)).called(1);
```

# UTILISATION DE MOCKING POUR LES TESTS DE PRÉFÉRENCES UTILISATEUR

- Mocking de `SharedPreferences` pour isoler les tests.
- Utilisation de `mockito` pour créer des objets mock.
- Vérification des interactions avec les objets mock.

```
final prefs = MockSharedPreferences();
when(prefs.getString('key')).thenReturn('value');
expect(prefs.getString('key'), 'value');
```

# MEILLEURES PRATIQUES POUR LES TESTS DE ROUTAGE

- Écrire des tests clairs et concis.
- Utiliser des mocks pour isoler les tests.
- Vérifier les routes et les transitions.
- Utiliser `pumpAndSettle` pour attendre les animations.
- Documenter les tests pour faciliter la maintenance.

# MEILLEURES PRATIQUES POUR LES TESTS DE PRÉFÉRENCES UTILISATEUR

- Écrire des tests clairs et concis.
- Utiliser des mocks pour isoler les tests.
- Vérifier les valeurs des préférences.
- Utiliser `setMockInitialValues` pour initialiser les préférences.
- Documenter les tests pour faciliter la maintenance.