Développement Android avec Kotlin

Cours - 09 - Navigation dans une application Android

Jordan Hiertz

Contact

hiertzjordan@gmail.com

jordan.hiertz@al-enterprise.com

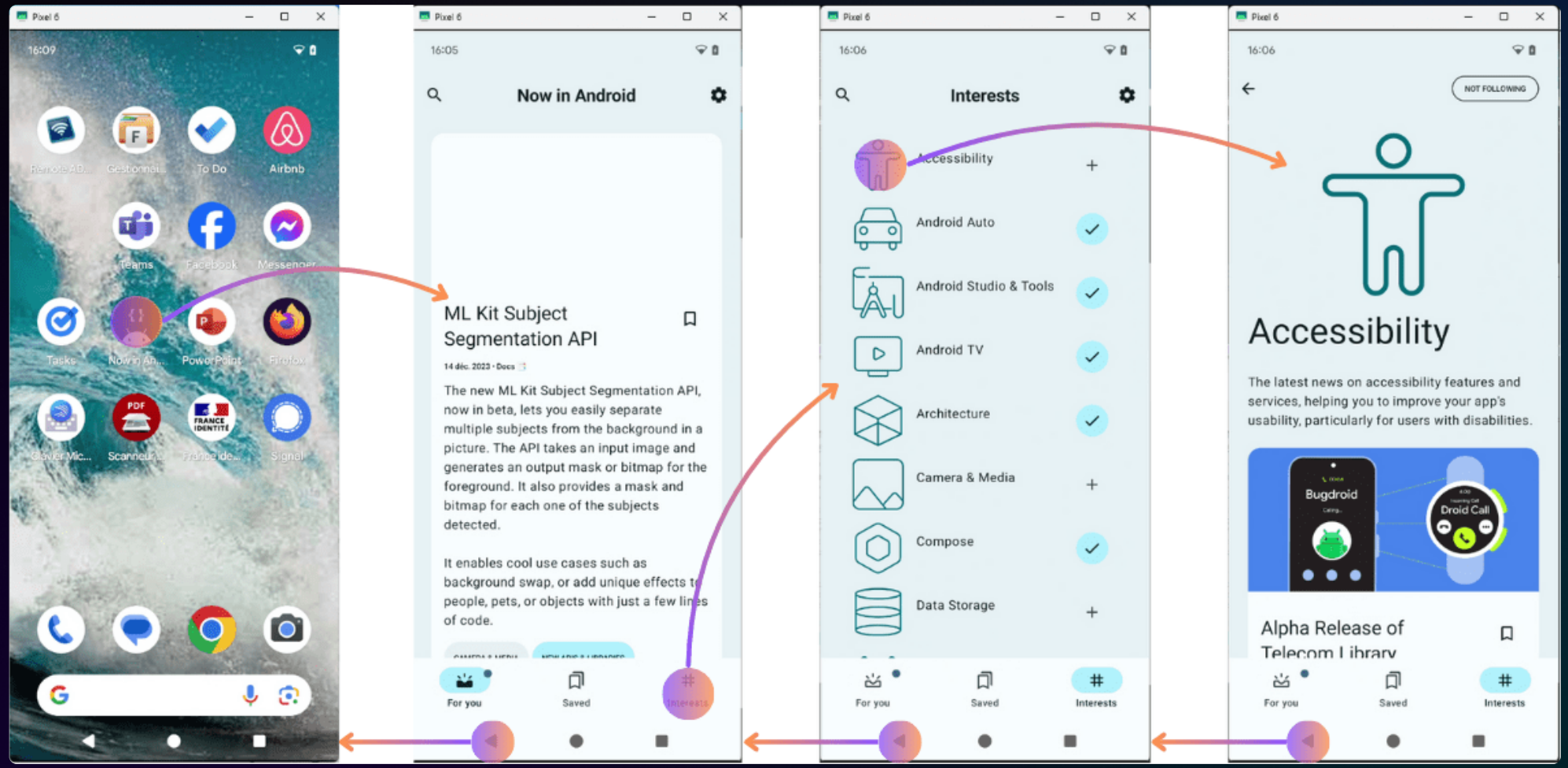
Principes de navigation - Introduction

- **Une navigation fluide et intuitive** améliore l'expérience utilisateur.
- **Des principes communs** garantissent une cohérence entre les applications Android.
- **Le composant Navigation** simplifie l'implémentation en appliquant ces principes par défaut.
- **Les utilisateurs retrouvent des schémas familiers**, facilitant leur adaptation aux différentes applications.

Principes de navigation - Destination de départ fixe

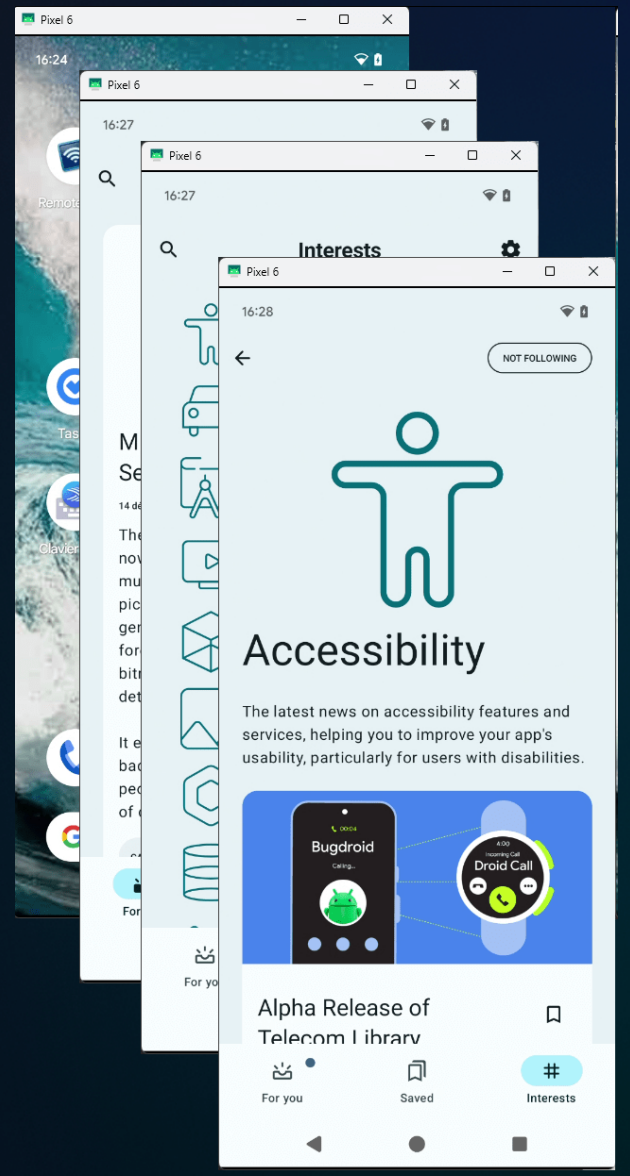
- **La destination de départ fixe** est le premier écran vu au lancement de l'application.
- Elle est également le dernier écran avant de revenir au lanceur d'applications via le bouton **"Retour"**.
- Cela garantit une **navigation cohérente** et une **expérience utilisateur fluide**.

Principes de navigation - Destination de départ fixe



Principes de navigation - Pile de destinations

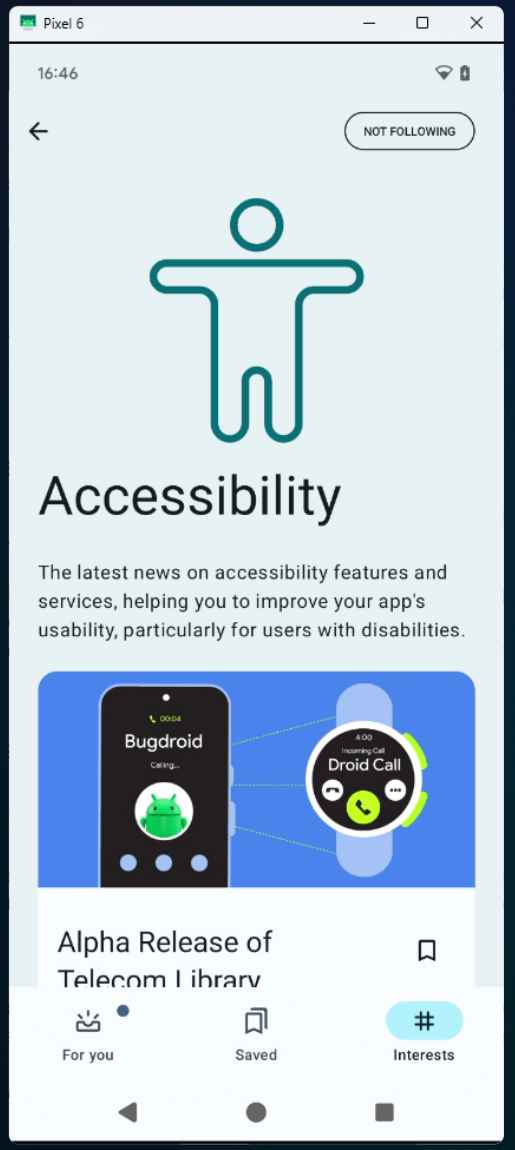
- Lancer l'application crée une nouvelle **"task" (tâche)** Android.
- **Une pile "Retour"** est créée et associée à cette task.
- **La destination de départ** reste toujours à la base de la pile.
- **L'écran actuel** est toujours au sommet de la pile.
- **Naviguer vers un nouvel écran** l'ajoute en haut de la pile.
- **Le bouton "Retour"** retire la destination du sommet et affiche la précédente.



Principes de navigation - Boutons "Retour" et "Haut"

- **Bouton "Haut" (barre d'application)**
  - Apparaît en **haut de l'écran** sous forme d'une flèche "<-".
  - Suit la hiérarchie de navigation de l'application.
  - **Ne quitte jamais l'application**.
  - N'apparaît pas sur la **destination de départ**.

- **Bouton "Retour" (système)**
  - Apparaît dans la **barre de navigation** (ou via un geste).
  - Permet de parcourir l'historique des écrans dans l'ordre inverse.
  - Peut **quitter l'application** si la pile est vide.



Principes de navigation - Liens profonds (Deep links)

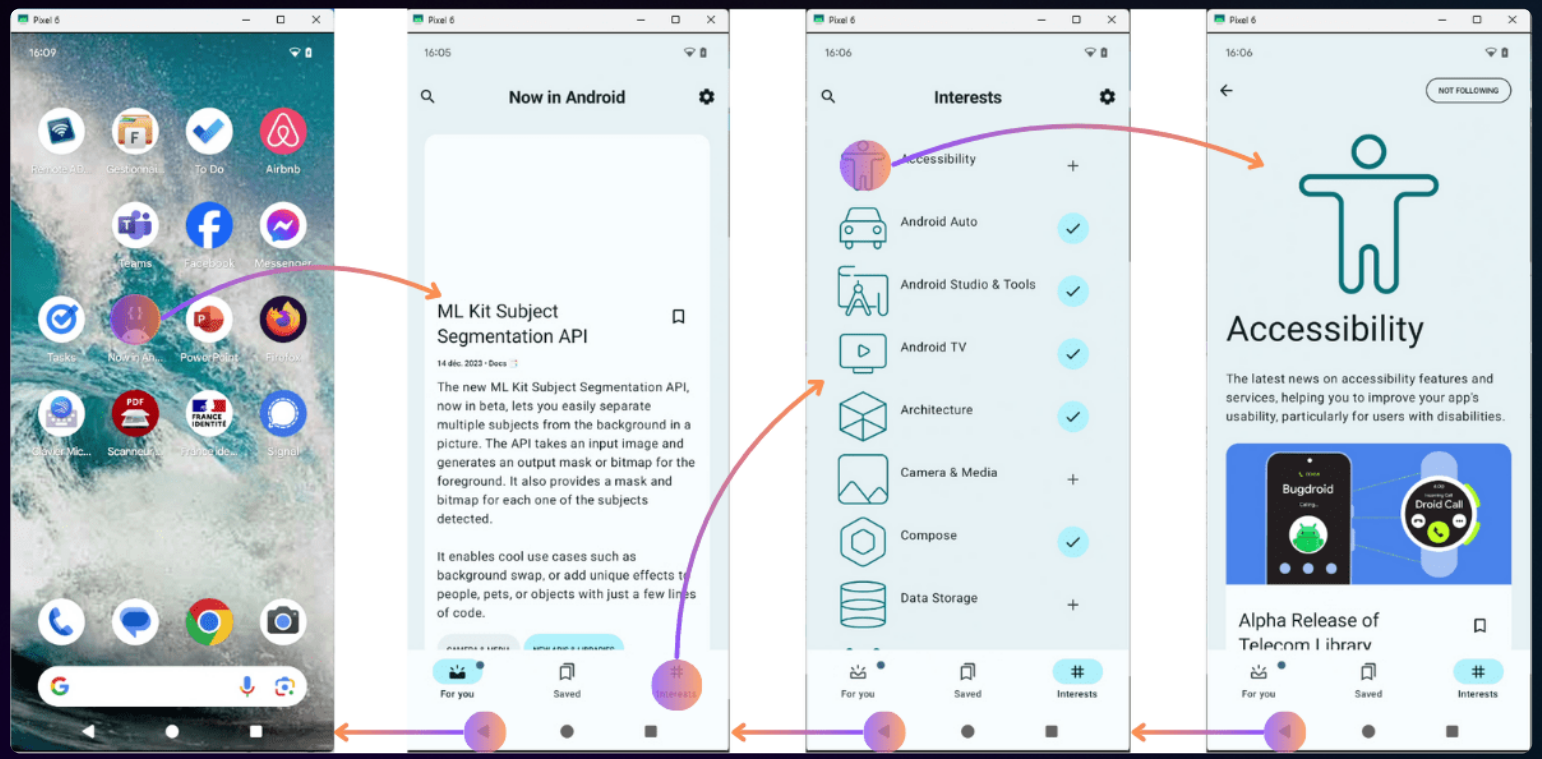
- Un **deep link** est une URL qui ouvre directement une destination spécifique dans votre application. Il peut être utilisé depuis :
  - Un navigateur web
  - Une autre application
  - Une notification

- Accès direct à une destination spécifique
  - Un lien profond permet d'accéder directement à une destination dans l'application.
  - Le bouton **"Haut"** permet alors de revenir à la **destination de départ** de l'application.

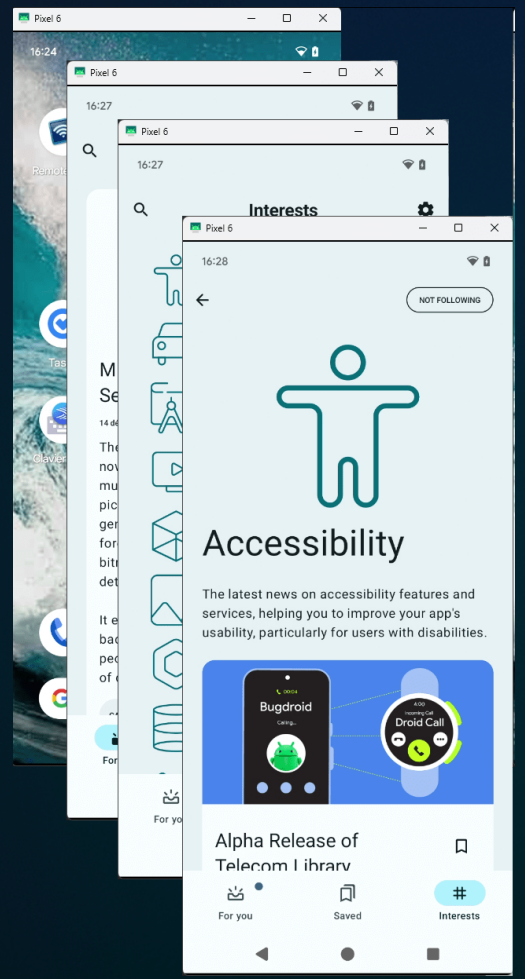
- **Impact sur la pile "Retour"**
  - Lorsqu'un lien profond est utilisé, la pile "Retour" existante est **remplacée** par une nouvelle pile contenant uniquement la destination cible et ses parents hiérarchiques.

Principes de navigation - Liens profonds

Navigation de l'application

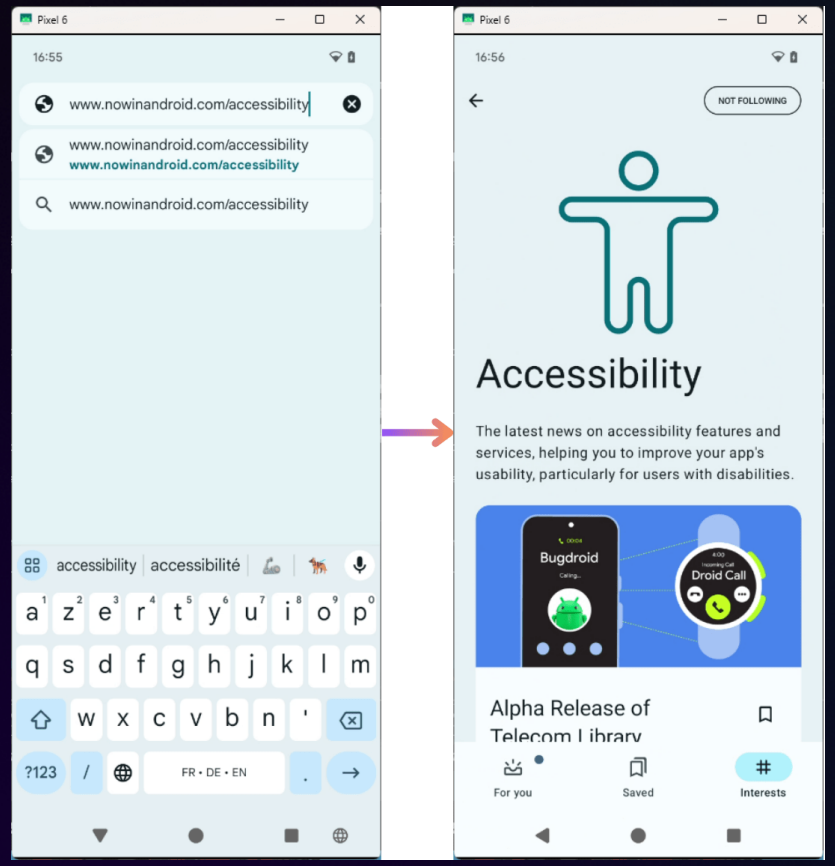


Résultat de la pile retour

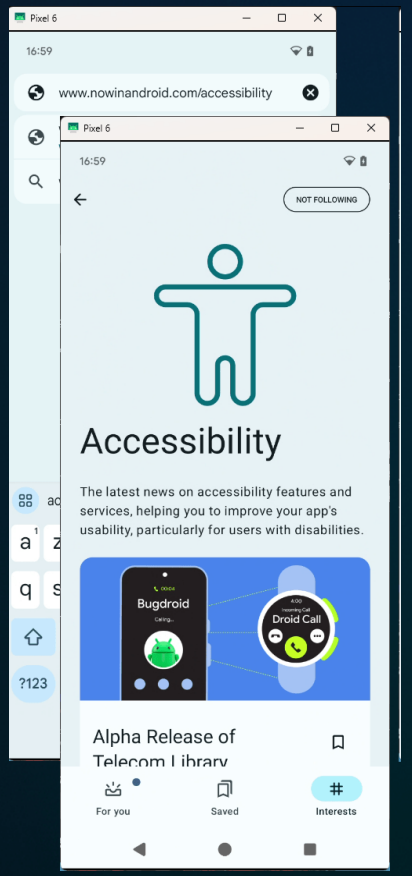


Principes de navigation - Liens profonds

Navigation de l'application



Résultat de la pile retour



Navigation dans une architecture multi-activité

- **Gestion manuelle avec Intent et Activity**
  - Chaque écran était une **Activity** distincte.
  - La navigation se faisait via des **Intents** explicites ou implicites.
  - La gestion de la pile "Retour" était manuelle et devenait complexe avec plusieurs **Activités** ouvertes.

- **Inconvénients d'une architecture multi-activité**
  - Chaque **Activity** était indépendante, nécessitant de gérer son propre **état**, ses **ressources**, et son **cycle de vie**.
  - Le passage d’informations entre **Activités** était fastidieux et souvent source d’erreurs.
  - Créer une nouvelle **Activity** était coûteux en termes de ressources.
  - L'architecture multi-activité est devenue **obsolète** avec l'émergence des **bibliothèques modernes**.

Navigation dans une architecture multi-activité

```
// Dans MainActivity, lorsque l'utilisateur clique sur un bouton, on lance DetailActivity
val button = findViewById<Button>(R.id.button)
button.setOnClickListener {
    val intent = Intent(this, DetailActivity::class.java)

    intent.putExtra("EXTRA_MESSAGE", "Détails sur l'élément")

    startActivity(intent)
}
```

```
class DetailActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_detail)

        // Récupération des données passées depuis MainActivity
        val message = intent.getStringExtra("EXTRA_MESSAGE")

        // Utilisation de la donnée récupérée
        val textView = findViewById<TextView>(R.id.textView)
        textView.text = message
    }
}
```

Navigation dans une architecture mono-activité

- Une seule **Activity** héberge plusieurs **fragments**, chacun représentant une section ou vue de l’application.

- La navigation se fait via des **transactions de fragments** (ajouter, remplacer ou supprimer des fragments dans l'Activity).

Avantages et limites de l'architecture mono-activité

- **Avantages :**
  - **Gestion centralisée** : L'état et les ressources sont gérés dans une seule **Activity**.
  - **Réutilisation des fragments** : Les fragments peuvent être utilisés dans différentes parties de l'application, ce qui évite la duplication de code.
  - **Optimisation des ressources** : Moins d'instances d'**Activity** créées, ce qui réduit la consommation mémoire.

- **Limites :**
  - **Complexité de gestion de la pile de fragments** : Gérer l'ordre des fragments et leurs transactions peut devenir difficile.
  - **Coordination avec l'Activity** : L'Activity doit gérer plusieurs fragments, ce qui peut compliquer la gestion de l'état à travers l'application.
  - **Cycle de vie des fragments** : Le cycle de vie des fragments est différent de celui de l'Activity et peut entraîner des erreurs ou des incohérences d'état.

Navigation dans une architecture mono-activité

```kotlin
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        ...

        if (savedInstanceState == null) {
            val listFragment = ListFragment()
            supportFragmentManager.beginTransaction()
                .replace(R.id.fragment_container, listFragment)  // Affichage de ListFragment
                .commit()
        }
    }
}
```

```kotlin
class ListFragment : Fragment() {

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        val button = view.findViewById<Button>(R.id.button_view_details)
        button.setOnClickListener {
            val detailFragment = DetailFragment()
            activity?.supportFragmentManager?.beginTransaction()
                ?.replace(R.id.fragment_container, detailFragment)  // Remplacer ListFragment par DetailFragment
                ?.addToBackStack(null)  // Ajout à la pile de fragments pour pouvoir revenir en arrière
                ?.commit()
        }
    }
}
```

Composant Navigation Android Jetpack

Le composant **Navigation** d'Android Jetpack permet de gérer la navigation dans une application Android de manière moderne et cohérente. Il comprend plusieurs éléments essentiels :

- **Bibliothèque Navigation** : Simplifie la gestion des écrans, des actions et de la pile de navigation.
- **Safe Args** : Un plug-in Gradle permettant de transmettre de manière sécurisée des données entre les destinations, en générant du code pour éviter les erreurs de typage.
- **Outils** : Intégration avec Android Studio pour faciliter la conception de la navigation et la gestion des destinations.

> **Note** : 🚀 *Le composant Navigation assure une expérience utilisateur cohérente et prévisible grâce au respect de l'ensemble des principes de navigation.*

L’hôte de navigation (NavHost)

L’**hôte de navigation** est le conteneur responsable de l’affichage des destinations et des transitions entre elles. Il existe deux types d’hôtes selon l’interface utilisée :

NavHostFragment (Views)

```xml
<androidx.navigation.fragment.NavHostFragment
    android:id="@+id/nav_host_fragment"
    android:name="androidx.navigation.fragment.NavHostFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:navGraph="@navigation/nav_graph" />
```

NavHost (Jetpack Compose)

```kotlin
NavHost(
    navController = navController,
    startDestination = "home"
) {
    composable("home") { HomeScreen() }
    composable("details") { DetailsScreen() }
}
```

L’hôte est un élément clé du Navigation Component, s’adaptant aussi bien aux interfaces basées sur Views qu’à celles construites avec Jetpack Compose.

Le graphique de Navigation (NavGraph)

Le **graphique de Navigation** est une structure qui définit les différentes destinations d'une application ainsi que les chemins de navigation entre elles. Il permet de centraliser la logique de navigation.

- **Décrit les destinations** : Activités, fragments (Views) ou écrans (Compose).
- **Définit les actions** : Transitions possibles entre les destinations.

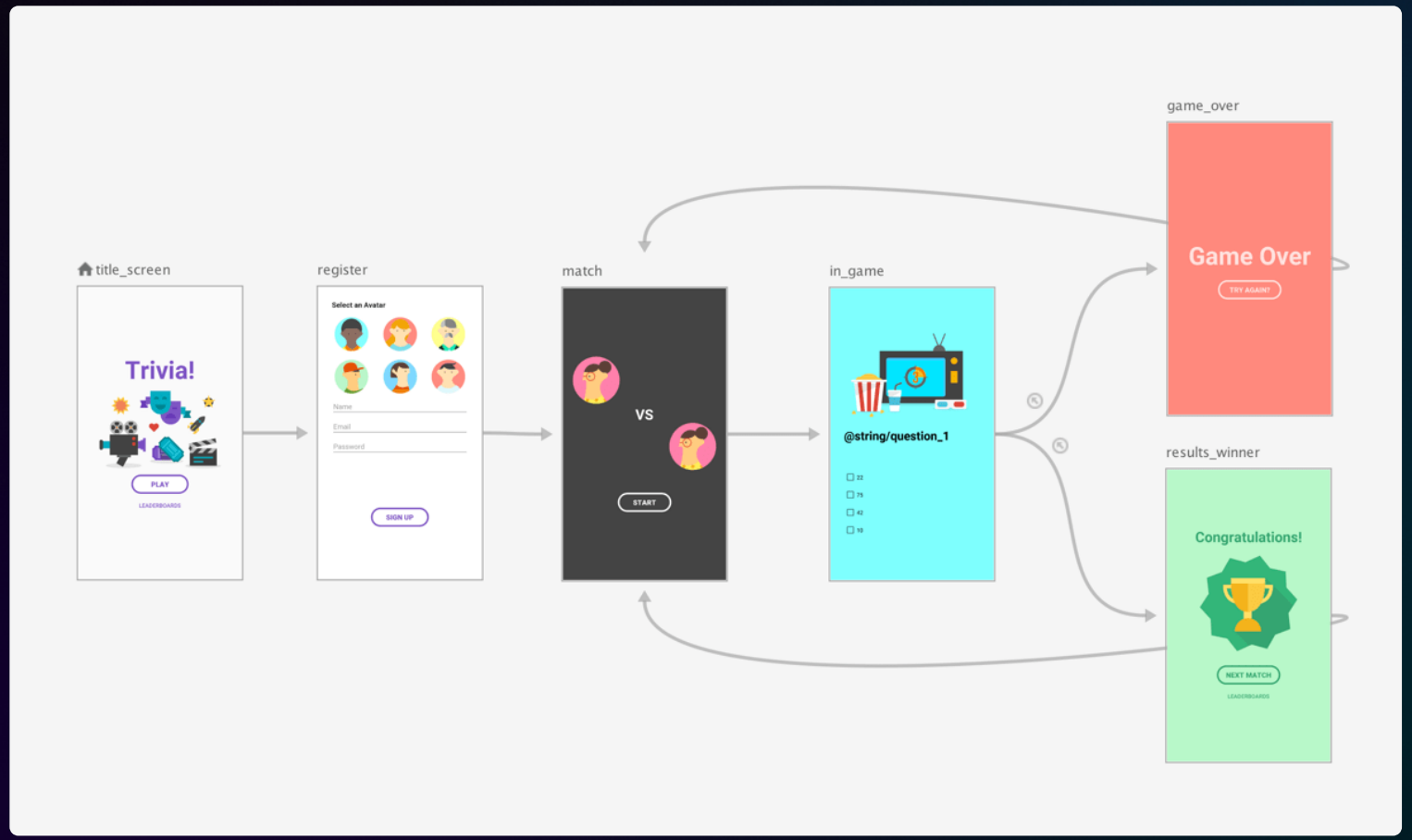
Le graphique de Navigation (NavGraph) - Views

Le graph est défini dans un fichier XML (ex: `res/navigation/nav_graph.xml`) :

```xml
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    app:startDestination="@id/homeFragment">

    <fragment
        android:id="@+id/homeFragment"
        android:name="com.example.HomeFragment">
        <action
            android:id="@+id/action_home_to_details"
            app:destination="@id/detailsFragment" />
    </fragment>

    <fragment
        android:id="@+id/detailsFragment"
        android:name="com.example.DetailsFragment" />
</navigation>
```

Le graphique de Navigation (NavGraph) - Views

Le graph est défini dans un fichier XML (ex: `res/navigation/nav_graph.xml`) :



Le graphique de Navigation (NavGraph) - Compose

Le `NavGraph` est défini via la fonction `NavHost` :

```
NavHost(
    navController = navController,
    startDestination = "home"
) {
    composable("home") { HomeScreen() }
    composable("details") { DetailsScreen() }
}
```

Le NavController : Piloter la Navigation

Le **NavController** est l'élément central du **Navigation Component** qui permet de gérer la navigation entre les destinations définies dans le **NavGraph**. Il est responsable des transitions et de la gestion de la pile de navigation.

- **Changer de destination** : Naviguer entre les écrans définis dans le `NavGraph`.
- **Gérer la pile de navigation** : Ajouter, supprimer ou revenir en arrière.
- **Passer des arguments** : Envoyer des données entre les destinations.
- **Supporter les Deep Links** : Ouvrir des destinations spécifiques via des URLs ou des Intent.

Utilisation du NavController - Views

Dans une activité utilisant `NavHostFragment` :

1️⃣ Récupération du NavController

```
1 val navController = findNavController(R.id.nav_host_fragment)
```

2️⃣ Navigation entre destinations

```
1 navController.navigate(R.id.detailsFragment)
```

3️⃣ Revenir en arrière

```
1 navController.popBackStack()
```

Utilisation du NavController - Compose

Dans Jetpack Compose :

1️⃣ Création du NavController

```
1 val navController = rememberNavController()
```

2️⃣ Navigation entre destinations

```
1 navController.navigate(route = DetailsScreen)
```

3️⃣ Revenir en arrière

```
1 navController.popBackStack()
```

Avantages et Fonctionnalités du Navigation Component

Le **Navigation Component** d'Android Jetpack offre une gestion moderne et simplifiée de la navigation, avec de nombreux avantages :

- **Animations et transitions** : Fournit des ressources standardisées pour appliquer facilement des animations et transitions entre les écrans.
- **Liens profonds (Deep Links)** : Gère les liens profonds pour rediriger l'utilisateur directement vers une destination spécifique.
- **Compatibilité avec les modèles d'UI** : Facilement intégré aux panneaux de navigation, à la barre de navigation inférieure et au menu latéral.
- **Sûreté du typage** : Transmission sécurisée de données entre destinations grâce à Safe Args.
- **Compatibilité avec ViewModel** : Étend la portée d'un ViewModel à tout un NavGraph pour partager des données entre écrans.
- **Gestion des Fragments** : Entièrement compatible avec les transactions de fragment.
- **Retour par balayage** : Prise en charge native du geste de balayage pour revenir en arrière.

Concevoir le graphique de navigation - Compose

Avant de commencer à utiliser le **Navigation Component** dans une application Jetpack Compose, il faut configurer l'environnement et ajouter les bibliothèques nécessaires.

```
[versions]
navigationCompose = "2.8.6"
kotlinxSerializationJson = "1.7.3"

[libraries]
#Jetpack Compose integration
androidx-navigation-compose = { module = "androidx.navigation:navigation-compose", version.ref = "navigationCompose" }

#JSON serialization library, works with the Kotlin serialization plugin
kotlinx-serialization-json = { module = "org.jetbrains.kotlinx:kotlinx-serialization-json", version.ref = "kotlinxSerializationJs

[plugins]
#Kotlin serialization plugin for type safe routes and navigation arguments
serialization = { id = "org.jetbrains.kotlin.plugin.serialization", version.ref = "kotlin" }
```

```
plugins {
    alias(libs.plugins.serialization)
}

dependencies {
    implementation(libs.androidx.navigation.compose)
    implementation(libs.kotlinx.serialization.json)
}
```

Concevoir le graphique de navigation - Compose

Voici un exemple d'une structure de projet recommandée pour gérer la navigation et organiser les écrans dans une application Jetpack Compose :

```
/ui
  ├── navigation/          # Contient la logique de navigation, y compris le NavGraph
  │     ├── NavGraph.kt    # Définition du graph de navigation
  │     ├── Destinations.kt # Routes et destinations de l'application
  ├── screens/             # Contient les différents écrans de l'application
  │     ├── HomeScreen.kt   # Écran d'accueil
  │     ├── DetailsScreen.kt # Écran de détails
  │     ├── ProfileScreen.kt # Écran de profil
  ├── theme/               # Personnalisation des styles et thèmes de l'application
  │     ├── Color.kt       # Définition des couleurs
  │     ├── Typography.kt # Définition de la typographie
  │     ├── Theme.kt       # Personnalisation du thème global
```

Concevoir le graphique de navigation - Compose

On utilise une classe sérialisable pour définir une navigation. Un itinéraire décrit comment on accède à cette destination et contient toutes les informations pour y parvenir.

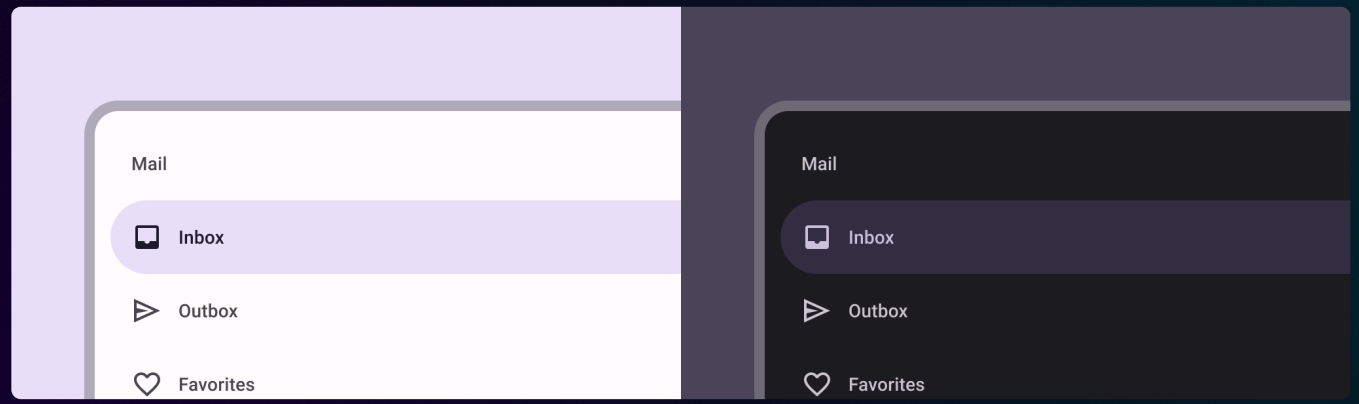
```kotlin
@Serializable
object Home

@Serializable
data class Details(val id: String)

@Serializable
object Profile


val navController = rememberNavController()

NavHost(navController = navController, startDestination = Profile) {
    composable<Home> { HomeScreen( /* ... */ ) }
    composable<Details> { DetailsScreen( /* ... */ ) }
    composable<Profile> { ProfileScreen( /* ... */ ) }
    // Add more destinations similarly.
}
```

Composant de navigation - Panneau de navigation

Un **menu coulissant** qui permet d'accéder à différentes sections de l'application. Ce composant est très utile pour naviguer rapidement entre les principales sections sans encombrer l'interface.

Fonctionnalités :

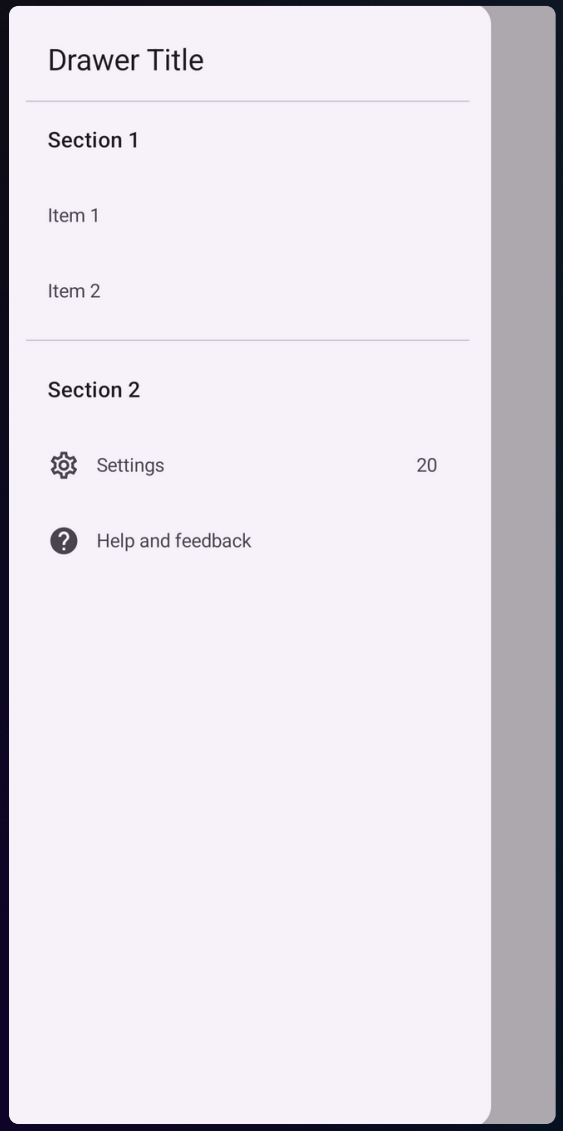
- **Affichage du menu** : Le menu peut être ouvert via une icône de type "hamburger" ou par un balayage depuis le côté.
- **Navigation rapide** : Permet de sauter d'une section à l'autre sans quitter l'écran principal.



Composant de navigation - Panneau de navigation

```kotlin
val drawerState = rememberDrawerState(initialValue = DrawerValue.Closed)
val scope = rememberCoroutineScope()

ModalNavigationDrawer(
    drawerContent = {
        ModalDrawerSheet {
            Text("Drawer title", modifier = Modifier.padding(16.dp))
            HorizontalDivider()
            NavigationDrawerItem(
                label = { Text(text = "Drawer Item") },
                selected = false,
                onClick = { /*TODO*/ }
            )
            // ...other drawer items
        }
    },
    gesturesEnabled = true
) {
    // Screen content
    ...
    onClick = {
        scope.launch {
            drawerState.apply {
                if (isClosed) open() else close()
            }
        }
    }
}
```

Composant de navigation - Panneau de navigation



Composant de navigation - Navigation Bar

La **Navigation Bar** (ou barre de navigation inférieure) permet aux utilisateurs de naviguer rapidement entre les sections principales de l'application. Elle est souvent utilisée pour des applications avec un nombre limité de sections principales.

Fonctionnalités :

- **Icônes de navigation** : Chaque élément représente une destination spécifique dans l'application.
- **Indication de l'écran actif** : L'élément actif est mis en surbrillance pour indiquer où l'utilisateur se trouve dans l'application.

Composant de navigation - Navigation Bar

```
var selectedItem by remember { mutableIntStateOf(0) }
val items = listOf("Songs", "Artists", "Playlists")
val selectedIcons = listOf(Icons.Filled.Home, Icons.Filled.Favorite, Icons.Filled.Star)
val unselectedIcons = listOf(Icons.Outlined.Home, Icons.Outlined.FavoriteBorder, Icons.Outlined.StarBorder)

NavigationBar {
    items.forEachIndexed { index, item ->
        NavigationBarItem(
            icon = {
                Icon(
                    if (selectedItem == index) selectedIcons[index] else unselectedIcons[index],
                    contentDescription = item
                )
            },
            label = { Text(item) },
            selected = selectedItem == index,
            onClick = { selectedItem = index }
        )
    }
}
```

Composant de navigation - Barres d'application supérieure

Les **barres d'application supérieure** sont des éléments essentiels de l'interface utilisateur qui permettent aux utilisateurs d'accéder rapidement aux principales fonctionnalités de l'application, comme le titre de l'écran, les actions spécifiques ou la navigation.

Fonctionnalités :

- **Titre de l'écran** : Affiche le nom de l'écran ou de la section actuelle.
- **Icônes d'action** : Permet d'afficher des actions comme la recherche, les paramètres, ou d'autres actions spécifiques à l'écran.
- **Navigation** : Intègre souvent un bouton hamburger ou une flèche pour la navigation entre les écrans.

Composant de navigation - Barres d'application supérieure

```
@Composable
fun HomeScreen() {
    Scaffold(
        topBar = {
            TopAppBar(
                colors = TopAppBarDefaults.topAppBarColors(
                    containerColor = MaterialTheme.colorScheme.primaryContainer,
                    titleContentColor = MaterialTheme.colorScheme.primary,
                ),
                title = {
                    Text("Small Top App Bar")
                },
                navigationIcon = {
                    IconButton(onClick = { /* do something */ }) {
                        Icon(
                            imageVector = Icons.Filled.Menu,
                            contentDescription = "Localized description"
                        )
                    }
                },
            )
        },
    ) { innerPadding ->
        ScrollContent(innerPadding)
    }
}
```

Conclusion

La gestion de la navigation est un aspect crucial du développement d'applications Android. Grâce au **Navigation Component** de Jetpack, vous avez à votre disposition un outil moderne, flexible et robuste pour gérer la navigation entre vos écrans.

Principaux points à retenir :

- **Simplification de la gestion des écrans** : Le `NavController`, `NavHost` et le `NavGraph` facilitent la gestion des destinations et de la pile de navigation.
- **Sécurité du typage** : L'utilisation d'objets sérialisables pour définir les itinéraires assure une navigation sécurisée et évite les erreurs de typage.
- **Support des fonctionnalités avancées** : Le composant Navigation prend en charge des fonctionnalités comme les animations, les liens profonds, et la gestion des ViewModels pour partager des données entre les écrans.

Conclusion

Le **Navigation Component** et **Jetpack Compose** offrent une manière cohérente et moderne de gérer la navigation dans vos applications Android. En suivant les bonnes pratiques et en structurant bien votre projet, vous pouvez créer une expérience utilisateur fluide et prévisible, tout en simplifiant le code et en améliorant la maintenabilité de vos applications.