Guía de Laboratorio 1

Navegación Tipada en Flutter

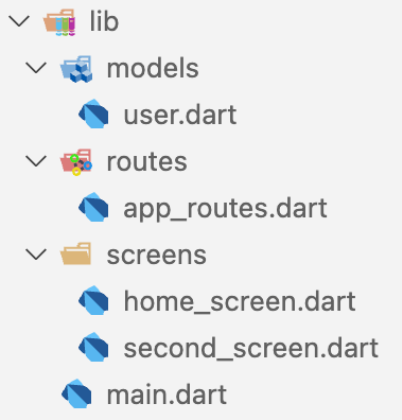
a) Objetivo del laboratorio

- Aprender a estructurar una aplicación Flutter que utilice rutas tipadas y personalizadas, centralizando la navegación en una clase AppRoutes.

Este enfoque mejora la organización, evita errores de rutas mal escritas y permite manejar parámetros o resultados de manera controlada.

b) Estructura del proyecto

Empieza creando la siguiente estructura de carpetas y archivos en tu proyecto.



c) Codificación

A continuación, listaremos el código de cada uno de los archivos que componen el proyecto, explicando la funcionalidad de los mismos. No dudes en preguntar si no entiendes alguna parte de la explicación.

Archivo: main.dart

```
import 'package:flutter/material.dart';
import 'package:navegacion_app/routes/app_routes.dart';

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

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
```

```
      title: 'Navegación Tipada',
      theme: ThemeData(
        colorSchemeSeed: Colors.indigo,
        brightness: Brightness.light,
        useMaterial3: false,
      ),
      debugShowCheckedModeBanner: false,
      onGenerateRoute: AppRoutes.onGenerateRoute,
      initialRoute: AppRoutes.home,
    );
  }
}
```

1. Importaciones del proyecto

```
import 'package:flutter/material.dart';
import 'package:navegacion_app/routes/app_routes.dart';
```

Explicación

- flutter/material.dart:
  Importa el **conjunto de widgets de Material Design**, que incluye componentes como Scaffold, AppBar, ElevatedButton, etc. Es la base visual y funcional de la app.
- navegacion_app/routes/app_routes.dart:
  Importa el **archivo donde están definidas las rutas** (por ejemplo: home, details, about), así como la función onGenerateRoute. En este laboratorio, esta separación cumple el principio de **responsabilidad única**, manteniendo el main.dart limpio y legible.

2. Función principal del programa

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

Explicación

- main() es el **punto de entrada** de toda aplicación Flutter.
- runApp() recibe el widget raíz de la aplicación (MyApp), que se monta sobre el árbol de widgets.
- Se usa const para **optimizar rendimiento**, ya que MyApp no tiene variables mutables.

En este punto se inicializa el motor gráfico de Flutter, se carga la app y se ejecuta el método build() de MyApp.

3. Clase principal MyApp

```
class MyApp extends StatelessWidget {
  const MyApp({super.key});
```

Explicación

- MyApp extiende de StatelessWidget, lo que significa que **no cambia su estado** durante la ejecución. (Perfecto para el widget raíz de una aplicación).
- {super.key} permite pasar una clave única si Flutter necesita optimizar la reconstrucción del widget.

4. Método build() de MyApp

```
@override
Widget build(BuildContext context) {
  return MaterialApp(
    title: 'Navegación Tipada',
```

Explicación

- build() devuelve un MaterialApp, que es el **contenedor principal** de toda app Flutter con estilo Material Design.
- El parámetro title define el **nombre de la app** que puede aparecer en tareas recientes o encabezados del sistema.

5. Configuración de tema

```
    theme: ThemeData(
      colorSchemeSeed: Colors.indigo,
      brightness: Brightness.light,
      useMaterial3: false,
    ),
```

Explicación

- ThemeData controla los **colores, fuentes y estilos globales** de la app.
- colorSchemeSeed: genera automáticamente una paleta de colores basándose en un color principal (en este caso, **índigo**).
- brightness: Brightness.light: usa el tema claro.
- useMaterial3: false: indica que se usará **Material Design 2**, útil para mantener compatibilidad con versiones anteriores.

En un laboratorio más avanzado podrías activar useMaterial3: true para ver cómo cambian los estilos.

6. Configuración de depuración

```
    debugShowCheckedModeBanner: false,
```

Explicación

- Oculta la etiqueta "**DEBUG**" que aparece por defecto en la esquina superior derecha cuando ejecutas la app en modo desarrollo.

7. Definición de rutas

```
      onGenerateRoute: AppRoutes.onGenerateRoute,
      initialRoute: AppRoutes.home,
```

Explicación

- onGenerateRoute:
  Se refiere a una función **centralizada** (definida en app_routes.dart) que decide **qué pantalla mostrar** cuando se invoca una ruta con Navigator.pushNamed. Este enfoque permite crear rutas dinámicas, pasar argumentos y manejar errores de navegación desde un solo punto.
- initialRoute:
  Define cuál será la **pantalla inicial** de la app (por ejemplo AppRoutes.home).

8. Cierre del método

```
    );
  }
}
```

Explicación

Cierra la estructura de MaterialApp y del método build(). A partir de este punto, toda la navegación será gestionada por el sistema de rutas definido en AppRoutes.

Archivo: user.dart

```dart
import 'package:flutter/foundation.dart';

@immutable
class User {
  final String nombre;
  final int edad;
  final String email;

  const User({required this.nombre, required this.edad, required this.email});

  Map<String, Object> toMap() => {
    'nombre': nombre,
    'edad': edad,
    'email': email,
  };

  factory User.fromMap(Map<String, Object?> map) => User(
    nombre: (map['nombre'] as String?) ?? '',
    edad: (map['edad'] as int?) ?? 0,
    email: (map['email'] as String?) ?? '',
  );
```

```
}
```

1. Importación de la librería de Flutter

```
import 'package:flutter/foundation.dart';
```

Explicación

- foundation.dart es parte del **núcleo del SDK de Flutter** e incluye clases fundamentales y anotaciones útiles.
- En este caso se importa para usar:
    - **@immutable**, que indica que los campos de la clase **no deben cambiar** después de su creación.

2. Declaración de la clase User

```
@immutable
class User {
  final String nombre;
  final int edad;
  final String email;
```

Explicación

- @immutable:
  Esta anotación le dice al analizador de Flutter (analyzer) que la clase debe ser **inmutable**. Es decir, una vez creado el objeto User, **no se pueden modificar sus valores**.
- final:
  Garantiza que cada propiedad (nombre, edad, email) se asigne **una sola vez** (al crear el objeto).

Importancia en Flutter:

Las clases inmutables son ideales para widgets y modelos porque **facilitan el control del estado**, evitan efectos secundarios y mejoran el rendimiento al reconstruir interfaces.

3. Constructor constante

```
  const User({required this.nombre, required this.edad, required this.email});
```

Explicación

- const:
  Permite crear **instancias constantes** del objeto, lo que reduce el consumo de memoria y acelera la ejecución cuando se usa con valores fijos.
- required:
  Indica que todos los parámetros son **obligatorios** al instanciar la clase. Por ejemplo:

- const user = User(nombre: 'Ana', edad: 25, email: 'ana@gmail.com');

Este patrón es muy útil cuando se pasan objetos entre rutas, ya que Flutter puede optimizar la comparación y reconstrucción de widgets.

4. Método toMap()

```
Map<String, Object> toMap() => {
  'nombre': nombre,
  'edad': edad,
  'email': email,
};
```

Explicación

- Convierte la instancia User a un **mapa de tipo clave–valor (Map)**. Esto permite:
  - Enviar el objeto como **argumento de navegación** (Navigator.pushNamed).
  - Serializarlo fácilmente para guardar en una base de datos o enviar por red.

```
Ejemplo de uso:
final user = User(nombre: 'Juan', edad: 30, email: 'juan@mail.com');
final data = user.toMap();
// Resultado: {'nombre': 'Juan', 'edad': 30, 'email': 'juan@mail.com'}
```

5. Fábrica fromMap()

```
factory User.fromMap(Map<String, Object?> map) => User(
  nombre: (map['nombre'] as String?) ?? '',
  edad: (map['edad'] as int?) ?? 0,
  email: (map['email'] as String?) ?? '',
);
```

Explicación

- **factory**:
  Define un **constructor especial** que puede devolver una instancia existente o una nueva según la lógica interna. Aquí se usa para **reconstruir un User a partir de un Map**.
- **Conversión segura (as Tipo? ?? valorPredeterminado)**: Se convierte cada campo con validación nula:
  - Si map['nombre'] no existe o no es String, devuelve ''.
  - Si map['edad'] no existe o no es int, devuelve 0.
  - Si map['email'] no existe o no es String, devuelve ''.

Esto evita excepciones (null errors) al recibir datos incompletos o al navegar con argumentos faltantes.

Archivo: app_routes.dart

```dart
import 'package:flutter/material.dart';
import 'package:navegacion_app/screens/home_screen.dart';
import 'package:navegacion_app/screens/second_screen.dart';
import 'package:navegacion_app/models/user.dart';

/// Clase que centraliza las rutas de la aplicación
abstract class AppRoutes {
  static const String home = '/';
  static const String userDetails = '/user-details';

  static const _errorMsg =
      'Los argumentos para /user-details deben ser de tipo UserDetailsArgs.';

  /// Método que genera las rutas según su nombre
  static Route<dynamic> onGenerateRoute(RouteSettings settings) {
    switch (settings.name) {
      case home:
        // Ruta principal
        return MaterialPageRoute<dynamic>(builder: (_) => const HomeScreen());

      case userDetails:
        final args = settings.arguments;
        if (args is! UserDetailsArgs) {
          return _errorRoute(_errorMsg);
        }

        // 🔷 Ruta tipada: devuelve un String como resultado
        return MaterialPageRoute<String>(
          builder: (_) => SecondScreen(titulo: args.titulo, user: args.user),
        );

      default:
        return _errorRoute('Ruta no encontrada: ${settings.name}');
    }
  }

  /// Pantalla de error genérica
  static Route<dynamic> _errorRoute(String message) {
    return MaterialPageRoute<dynamic>(
      builder: (_) => Scaffold(
        appBar: AppBar(title: const Text('Error de navegación')),
        body: Center(
          child: Padding(
            padding: const EdgeInsets.all(16),
            child: Text(
              message,
```

```
              textAlign: TextAlign.center,
              style: const TextStyle(fontSize: 16),
            ),
          ),
        ),
      ),
    );
  }
}

/// Argumentos tipados para la ruta de detalles del usuario
class UserDetailsArgs {
  final String titulo;
  final User user;

  const UserDetailsArgs({required this.titulo, required this.user});
}
```

1. Importaciones

```
import 'package:flutter/material.dart';
import 'package:navegacion_app/screens/home_screen.dart';
import 'package:navegacion_app/screens/second_screen.dart';
import 'package:navegacion_app/models/user.dart';
```

- Traes **Flutter** y las pantallas destino (HomeScreen, SecondScreen), además del **modelo** User.
- Son los "ingredientes" que tu generador de rutas usará para construir cada pantalla.

2. Clase centralizadora de rutas

```
abstract class AppRoutes {
  static const String home = '/';
  static const String userDetails = '/user-details';

  static const _errorMsg =
      'Los argumentos para /user-details deben ser de tipo UserDetailsArgs.';
```

- abstract class + static const: patrón para **centralizar nombres de rutas** (evita strings duplicadas y typos).
- _errorMsg: mensaje estándar cuando los argumentos no tienen el tipo correcto.

3. Generador de rutas: el corazón de tu navegación

```
  static Route<dynamic> onGenerateRoute(RouteSettings settings) {
    switch (settings.name) {
      case home:
```

```
        return MaterialPageRoute<dynamic>(builder: (_) => const HomeScreen());

      case userDetails:
        final args = settings.arguments;
        if (args is! UserDetailsArgs) {
          return _errorRoute(_errorMsg);
        }

        // 🔷 Ruta tipada: devuelve un String como resultado
        return MaterialPageRoute<String>(
          builder: (_) => SecondScreen(titulo: args.titulo, user: args.user),
        );

      default:
        return _errorRoute('Ruta no encontrada: ${settings.name}');
    }
  }
```

¿Qué hace?

- Recibe un RouteSettings (nombre y argumentos) y **decide** qué Route crear.

Caso home

- Devuelve MaterialPageRoute<dynamic> con HomeScreen. *No esperas resultado al volver a home, por eso va bien dynamic.*

Caso userDetails

- **Valida** que settings.arguments sea de tipo **UserDetailsArgs** (tu DTO tipado).
  - Si **no** lo es manda a una ruta de **error** (*fail fast*).
- Si es correcto crea una MaterialPageRoute<String> hacia SecondScreen.
  - Ojo: **el genérico <String>** declara que esta ruta **puede devolver** un String al hacer Navigator.pop<String>(...) en la pantalla destino.

Esto es clave para evitar tu crash: Si vas a invocar esta ruta con Navigator.pushNamed<String>(...), entonces el **Route que devuelves debe ser Route<String>** (por eso MaterialPageRoute<String>). Si devolvieras MaterialPageRoute<dynamic>, Flutter intentaría castear a Route<String?> y obtendrías el error: type 'MaterialPageRoute<dynamic>' is not a subtype of type 'Route<String?>?'.

default

- Cubre cualquier nombre de ruta desconocido con una **pantalla de error**. Excelente para DX (developer experience).

4. Ruta de error reutilizable

```
  static Route<dynamic> _errorRoute(String message) {
    return MaterialPageRoute<dynamic>(
      builder: (_) => Scaffold(
        appBar: AppBar(title: const Text('Error de navegación')),
        body: Center(
          child: Padding(
            padding: const EdgeInsets.all(16),
            child: Text(
              message,
              textAlign: TextAlign.center,
              style: const TextStyle(fontSize: 16),
            ),
          ),
        ),
      ),
    );
  }
```

- Construye una pantalla simple con el **mensaje** recibido.
- Útil para:
    - Argumentos con tipo incorrecto.
    - Rutas inexistentes.

5. DTO de argumentos tipados (¡muy bien!)

```
class UserDetailsArgs {
  final String titulo;
  final User user;

  const UserDetailsArgs({required this.titulo, required this.user});
}
```

- Defines un "**paquete de argumentos**" con **tipos fuertes** (evita Maps anónimos).
- Ventajas:
    - Autocompletado, refactors seguros, y menos errores en runtime.
    - Documenta explícitamente **qué necesita** la ruta.

Archivo: home_screen.dart

```
import 'package:flutter/material.dart';
import 'package:navegacion_app/models/user.dart';
import 'package:navegacion_app/routes/app_routes.dart';
```

```dart
class HomeScreen extends StatelessWidget {
  const HomeScreen({super.key});

  Future<void> _irASegundaPantalla(BuildContext context) async {
    // Crear un usuario de ejemplo
    const user = User(nombre: 'Juan', edad: 25, email: 'juan@example.com');

    // Navegar a la segunda pantalla usando rutas con nombre
    final mensaje = await Navigator.of(context).pushNamed<String>(
      AppRoutes.userDetails,
      arguments: const UserDetailsArgs(
        titulo: 'Detalles del Usuario',
        user: user,
      ),
    );

    // Verificar que el widget siga montado después del await
    if (!context.mounted) return;

    // Mostrar el resultado devuelto desde la segunda pantalla
    if (mensaje != null && mensaje.isNotEmpty) {
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(content: Text(mensaje), duration: const Duration(seconds: 4)),
      );
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Pantalla Principal')),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            const Text(
              'Esta es la pantalla principal',
              style: TextStyle(fontSize: 18),
            ),
            const SizedBox(height: 20),
            ElevatedButton(
              onPressed: () => _irASegundaPantalla(context),
              child: const Text('Ir a Segunda Pantalla'),
            ),
          ],
        ),
      ),
```

```
    );
  }
}
```

1. Importaciones

```
import 'package:flutter/material.dart';
import 'package:navegacion_app/models/user.dart';
import 'package:navegacion_app/routes/app_routes.dart';
```

- **Flutter Material**: widgets base (Scaffold, AppBar, ElevatedButton, etc.).
- **Modelo User**: objeto de dominio que enviarás como argumento a la segunda pantalla.
- **AppRoutes**: tu central de rutas y el DTO UserDetailsArgs para argumentos tipados.

2. Declaración del widget

```
class HomeScreen extends StatelessWidget {
  const HomeScreen({super.key});
```

- Widget **sin estado** (ideal para una pantalla simple que solo navega y muestra UI).
- const optimiza rendimiento (instancia inmutable).

3. Método privado de navegación

```
Future<void> _irASegundaPantalla(BuildContext context) async {
  // Crear un usuario de ejemplo
  const user = User(nombre: 'Juan', edad: 25, email: 'juan@example.com');
```

- Método asíncrono que encapsula la lógica de navegación (mejor legibilidad y testabilidad).
- user es **const** porque User es inmutable y su constructor es const.

```
  // Navegar a la segunda pantalla usando rutas con nombre
  final mensaje = await Navigator.of(context).pushNamed<String>(
    AppRoutes.userDetails,
    arguments: const UserDetailsArgs(
      titulo: 'Detalles del Usuario',
      user: user,
    ),
  );
```

- Usas pushNamed<String> **importante**: declaras que **esperas** un String como resultado cuando la segunda pantalla haga Navigator.pop<String>(...).
- AppRoutes.userDetails coincide con el case userDetails en tu onGenerateRoute.
- **Argumentos tipados** con UserDetailsArgs:
  - Evitas Map anónimos y errores de clave/valor en runtime.

- onGenerateRoute ya valida is! UserDetailsArgs y, si no coincide, redirige a _errorRoute.

- Gracias a tu onGenerateRoute, la ruta construida es MaterialPageRoute<String> **coherente** con el genérico <String> del pushNamed. Esto evita el crash: type 'MaterialPageRoute<dynamic>' is not a subtype of type 'Route<String?>?'.

```
// Verificar que el widget siga montado después del await
  if (!context.mounted) return;
```

- Defensa extra tras un await: asegura que el **árbol de widgets** no se haya desmontado (por ejemplo, si el usuario dejó la pantalla).
- Evita usar ScaffoldMessenger.of(context) con un BuildContext inválido.

```
  // Mostrar el resultado devuelto desde la segunda pantalla
  if (mensaje != null && mensaje.isNotEmpty) {
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(content: Text(mensaje), duration: const Duration(seconds: 4)),
    );
  }
}
```

- Si la segunda pantalla devolvió un String, lo muestras en un **SnackBar**.
- Manejas null/vacío con un guard clause simple.

Contrato implícito con la segunda pantalla: En SecondScreen debes hacer algo como:
`Navigator.pop<String>(context, 'OK desde detalles');`
Si no haces pop<String> (o devuelves otro tipo), el await pushNamed<String> no recibirá el valor esperado.

4. UI de la pantalla

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: const Text('Pantalla Principal')),
    body: Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
          const Text(
            'Esta es la pantalla principal',
            style: TextStyle(fontSize: 18),
          ),
          const SizedBox(height: 20),
          ElevatedButton(
            onPressed: () => _irASegundaPantalla(context),
```

```
            child: const Text('Ir a Segunda Pantalla'),
          ),
        ],
      ),
    ),
  );
}
```

- Estructura básica con Scaffold (barra superior + cuerpo).
- Un ElevatedButton dispara la navegación llamando a tu método asíncrono.
- Mantienes la **lógica separada de la UI** (buena práctica).

Archivo: second_screen.dart

```dart
import 'package:flutter/material.dart';
import 'package:navegacion_app/models/user.dart';

class SecondScreen extends StatelessWidget {
  final String titulo;
  final User user;

  const SecondScreen({super.key, required this.titulo, required this.user});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text(titulo)),
      body: Padding(
        padding: const EdgeInsets.all(16),
        child: Column(
          crossAxisAlignment: CrossAxisAlignment.start,
          children: [
            Text('Nombre: ${user.nombre}'),
            Text('Edad: ${user.edad}'),
            Text('Email: ${user.email}'),
            const SizedBox(height: 24),
            Row(
              children: [
                ElevatedButton(
                  onPressed: () {
                    Navigator.pop<String>(
                      context,
                      'Usuario guardado correctamente',
                    );
                  },
                  child: const Text('Guardar y volver'),
                ),
```

```
                const SizedBox(width: 12),
                OutlinedButton(
                  onPressed: () => Navigator.pop<String>(context, ''),
                  child: const Text('Cancelar'),
                ),
              ],
            ),
          ],
        ),
      ),
    );
  }
}
```

1. Importaciones

```
import 'package:flutter/material.dart';
import 'package:navegacion_app/models/user.dart';
```

- material.dart para construir la UI con Scaffold, AppBar, Buttons, etc.
- User es tu **modelo de dominio** que llega desde la ruta con argumentos tipados.

2. Widget inmutable con argumentos requeridos

```
class SecondScreen extends StatelessWidget {
  final String titulo;
  final User user;

  const SecondScreen({super.key, required this.titulo, required this.user});
```

- StatelessWidget porque la vista **no administra estado interno**; muestra datos y devuelve un resultado al salir.
- required garantiza que la pantalla **no pueda construirse** sin titulo y user (coincide con tu UserDetailsArgs en AppRoutes).

Gracias a tu AppRoutes.onGenerateRoute, esta pantalla **siempre** se crea con titulo y user válidos; si no, se muestra la ruta de error.

3. Construcción de la UI

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: Text(titulo)),
```

```
    body: Padding(
      padding: const EdgeInsets.all(16),
      child: Column(
        crossAxisAlignment: CrossAxisAlignment.start,
        children: [
          Text('Nombre: ${user.nombre}'),
          Text('Edad: ${user.edad}'),
          Text('Email: ${user.email}'),
          const SizedBox(height: 24),
          Row(
            children: [
              ElevatedButton(
                onPressed: () {
                  Navigator.pop<String>(
                    context,
                    'Usuario guardado correctamente',
                  );
                },
                child: const Text('Guardar y volver'),
              ),
              const SizedBox(width: 12),
              OutlinedButton(
                onPressed: () => Navigator.pop<String>(context, ''),
                child: const Text('Cancelar'),
              ),
            ],
          ),
        ],
      ),
    ),
  );
}
```

Puntos clave

- **Muestra** datos del usuario recibidos: nombre, edad, email.
- Botonera de acciones:
  - **Guardar y volver** Navigator.pop<String>(context, 'Usuario guardado correctamente')
    - Devuelve un **String** al caller (coincide con pushNamed<String> y MaterialPageRoute<String>).
  - **Cancelar** Navigator.pop<String>(context, '')
    - Devuelve una **cadena vacía** como "no-op" (tu HomeScreen la filtra con isNotEmpty).