



Programación multimedia y dispositivos móviles.

Flutter: MovieDb App - APIs

Alumno: Damian Altamirano Ontiveros Rivero.

Índice

.....	1
0. Introducción.....	4
1. Librerías Principales	5
1.1 GetX (get)	5
1.2 Flutter SVG (flutter_svg)	5
1.3 HTTP (http).....	5
1.4 Fade Shimmer (fade_shimmer)	5
1.5 Flutter Launcher Icons.....	5
1.6 Versiones del proyecto.....	5
2. EndPoints - TMDB API.....	6
3. Proyecto TMBD.....	7
3.1 Main	8
3.2 Screens	9
3.2.1 Main.....	9
3.2.2 Home_Screen	11
3.2.3 Details Screen.....	13
3.2.4 People Search Screen	16
3.2.5 Favorite List Screen.....	19
3.3 Controllers	21
3.3.1 Bottom navigator controller	21
3.3.2 People Controller.....	22
3.3.3 People Search Controller	24
3.4 Widgets.....	25
3.4.1 Index Number	25
3.4.2 Person Infos	26
3.4.3 People tab builder.....	27
3.4.4 People top rated item	29
3.4.5 Search Box.....	31
3.5 Models.....	33
3.6 API.....	34
3.6.1 Api Credentials.....	34
3.6.2 Api Services	34
4. Ampliaciones.....	37
4.1 Ampliación 1.....	37
4.2 Ampliación 2 y 3.....	41
Recursos.....	43
Demos.....	43
Demo TMDB App principal (Actors)	43
Demo TMDB App Ampliación 1.....	43

Demo TMDB App Ampliación 2 y 3.....	43
Librerías	43
Get.....	43
flutter_svg	43
http.....	43
fade_shimmer.....	43
flutter_launcher_icons.....	43
APIs.....	43
TMDB API.....	43
Repositoryos GitHub	43
Repositorio Personal:	43
Repositorio Proyecto base TheMovieDb:.....	44

0. Introducción

Se nos ha proporcionado un [proyecto base](#) para desarrollar una aplicación capaz de visualizar y buscar actores de películas, así como tener una lista de seguimiento donde podremos guardar la información de estos y ver las películas donde han participado.

Para acceder a estos datos, usaremos la API opensource de [TheMoviedb](#), la cual nos permite acceder a datos actualizados de ratings de películas y actores en tiempo real. Partiendo de una base de código, iremos modificando y documentando cada módulo de nuestra aplicación.

1. Librerías Principales

1.1 GetX (get)

Es una de las librerías más completas para Flutter. Se utiliza principalmente para tres cosas: gestión de estado (actualizar la pantalla cuando cambian los datos), gestión de rutas (navegar entre pantallas sin usar el "Context") y gestión de dependencias. Es muy popular porque simplifica mucho el código y separa la lógica de negocio de la interfaz visual.

1.2 Flutter SVG (flutter_svg)

Por defecto, Flutter no tiene soporte nativo para archivos vectoriales .svg. Esta dependencia permite renderizar este tipo de imágenes, las cuales son ideales para iconos e ilustraciones porque no pierden calidad al cambiar de tamaño y pesan mucho menos que un .png o un .jpg.

1.3 HTTP (http)

Una librería que nos proporciona los desarrolladores de Dart para facilitar las peticiones http. Este paquete contiene un conjunto de funciones y clases de alto nivel que lo componen fácil de consumir recursos HTTP. Es multiplataforma (móvil, escritorio y navegador) y soporta múltiples implementaciones.

1.4 Fade Shimmer (fade_shimmer)

Se utiliza para mejorar la experiencia de usuario mientras la aplicación carga datos. Crea ese efecto de "esqueleto" animado (luces grises que se mueven) que ves en aplicaciones como Facebook o Instagram antes de que aparezca el contenido real, evitando que el usuario vea una pantalla vacía o un círculo de carga aburrido.

1.5 Flutter Launcher Icons

Es una herramienta de desarrollo que automatiza la creación de los **iconos de la aplicación** tanto para Android como para iOS. En lugar de redimensionar manualmente una imagen a 20 tamaños diferentes, esta librería toma un archivo fuente y genera automáticamente todos los formatos necesarios por ti.

1.6 Versiones del proyecto

Estas son las versiones que hemos usado de cada una de las librerías ya mencionadas para este proyecto:

```
dependencies:  
  flutter:  
    sdk: flutter  
  
  get: ^4.7.3  
  flutter_svg: ^2.2.3  
  http: ^1.6.0  
  fade_shimmer: ^2.4.0  
  flutter_launcher_icons: ^0.14.1
```

2. EndPoints - TMDB API.

Estos los endpoints oficiales que usaremos para construir nuestros servicios en [ApiService](#):

Person Popular: Este servicio devuelve una lista global de los actores y profesionales del cine que tienen el mayor índice de relevancia en el momento actual. Se basa en una combinación de factores como búsquedas de usuarios, visualizaciones de sus perfiles y actividad reciente en la industria, proporcionando una visión general de las figuras más reconocidas.

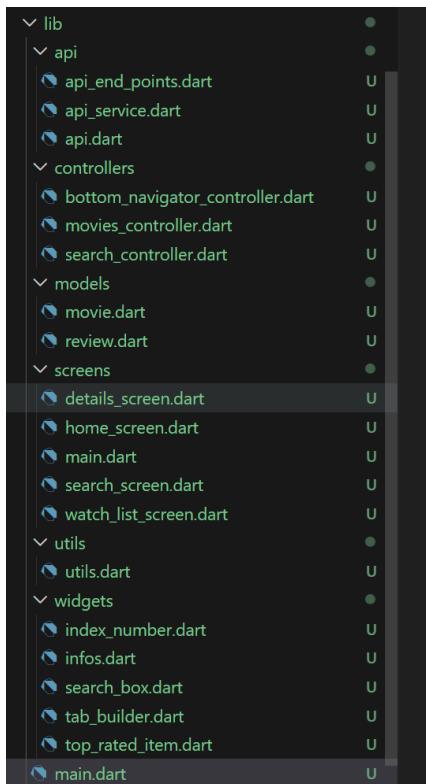
Person Details: Es el endpoint de consulta profunda. A diferencia de las listas generales que solo ofrecen nombres y fotos, este servicio permite acceder a la ficha técnica completa de un individuo. Proporciona datos personales valiosos como su biografía completa, lugar de nacimiento oficial, fecha de nacimiento, redes sociales asociadas y otros detalles biográficos específicos que definen su trayectoria.

Trending People: Este recurso se especializa en identificar picos de popularidad en ventanas de tiempo específicas (diarias o semanales). Es ideal para detectar qué personalidades están "en boca de todos" debido a estrenos recientes, noticias o eventos globales, diferenciándose del ranking de popularidad general por su naturaleza inmediata y volátil.

Search Person: Es la herramienta de consulta directa dentro de la base de datos de TMDB. Este servicio escanea miles de registros para encontrar coincidencias exactas o parciales con el nombre de un actor. Está diseñado para resolver búsquedas específicas de los usuarios, devolviendo perfiles que encajen con los criterios de búsqueda textual.

3. Proyecto TMBD

Una tenemos claras las herramientas que usaremos para construir nuestra aplicación, empezaremos viendo la estructura y la distribución de módulos de este:



A lo largo de esta documentación iremos viendo cada uno de estos módulos y analizándolos. Además, comentaremos los cambios que iremos realizando partiendo de la base ya establecida para llegar a nuestro resultado final, ademas de dejar la puerta abierta a nuevas funcionalidades futuras.

Si se desea ver el funcionamiento de la aplicación en tiempo real, ir al [Demo](#) para verlo.

3.1 Main

Partimos importando las siguientes librerías a nuestro código:

```
import 'package:flutter/material.dart';
import 'package:flutter/services.dart';
import 'package:get/get.dart';
import 'package:movies/screens/main.dart';
```

Una vez hecho esto, declararemos la raíz de nuestro proyecto:

```
// Entry point of the application. We run the root widget: MyApp
Run | Debug | Profile
void main() {
  runApp(const MyApp());
}
```

Y empezaremos a definir esta clase, definiendo en el proceso su constructor:

```
class MyApp extends StatelessWidget {
  // Constant constructor. Passes the optional key to the parent class.
  const MyApp({super.key});
```

para definir temas generales de temas y color, sobrescribiremos el método base de la clase **build**. Ya que este método es llamado por la aplicación cada vez que se crea la interfaz UI, definiremos dentro de este el color de la barra de estado de nuestra aplicación:

```
// Override the build method from StatelessWidget.
// It receives the BuildContext and returns the widget tree.
@Override
Widget build(BuildContext context) {
  // Set the default system status bar color.
  SystemChrome.setSystemUIOverlayStyle(
    const SystemUiOverlayStyle(
      statusBarColor: Color(0xFF242A32),
    ), // SystemUiOverlayStyle
  );
}
```

Ahora, haciendo de uso de una librería muy popular llamada **GetX**, construiremos el árbol de widgets que nos retornará el método partiendo de esa base. **GetX** nos permitirá tener una mayor gestión de los estados de nuestra aplicación, poder navegar entre pantallas de manera más eficiente y la posibilidad de construir servicios capaces de ser llamados de manera más simple.

```
// Use GetX's GetMaterialApp as the root widget of the application.
return GetMaterialApp(
  // Remove the debug banner.
  debugShowCheckedModeBanner: false,
  // Global theme configuration: colors and text styles.
  theme: ThemeData(
    scaffoldBackgroundColor: const Color(0xFF242A32),
    textTheme: const TextTheme(
      bodyLarge: TextStyle(
        color: Colors.white,
        fontFamily: 'Poppins',
      ), // TextStyle
      bodyMedium: TextStyle(
        color: Colors.white,
        fontFamily: 'Poppins',
      ), // TextStyle
    ), // TextTheme
  ), // ThemeData
)
```

Una vez hecho esto, tendremos la base de nuestra aplicación hecha. Ahora, nos centraremos en la pantalla principal de nuestra aplicación.

```
// Initial screen of the application.
home: Main(),
```

3.2 Screens

Dentro de este módulo configuraremos las diferentes pantallas de nuestra aplicación. Veremos su funcionamiento, así como las dependencias con otros módulos y sus aplicaciones.

3.2.1 Main

Partimos importando las siguientes librerías a nuestro código:

```
import 'package:flutter/material.dart';
import 'package:flutter_svg/flutter_svg.dart';
import 'package:get/get.dart';
import 'package:movies/controllers/bottom_navigator_controller.dart';
```

En este módulo nos encargaremos de configurar la barra de navegación mediante GetX, el cual estará situada en la parte inferior de la aplicación. Declararemos la clase Main, y su constructor, así como la creación de una nueva instancia del objeto **BottomNavigatorController**, en el cual llamaremos al método get de la librería **GetX** para inyectarlo en memoria.

```
class Main extends StatelessWidget {
    Main({super.key});
    final BottomNavigatorController controller = Get.put(BottomNavigatorController());
```

Ahora, sobreescribiremos el método **build**, dentro de este declararemos un objeto **Obx** el cual nos es mas que un controlador que envolverá al navegador, para que de manera automática y sin la necesidad de hacer un **setState**, poder cambiar de pantallas cuando cambie el índice en el navegador cada vez que nuestra aplicación cree la UI.

```
@override
Widget build(BuildContext context) {
    return Obx(
        () => GestureDetector(
            onTap: () {
                FocusScope.of(context).unfocus();
            },
            child: Scaffold(
                body: SafeArea(
                    child: IndexedStack(
                        index: controller.index.value,
                        children: Get.find<BottomNavigatorController>().screens,
                    ), // IndexedStack
                ), // SafeArea
            )
        )
    );
}
```

Siguiendo con las configuraciones, especificaremos el comportamiento del evento **onTap()**, el cual, mediante los iconos que ya tengamos de opciones, al presionar sobre una de estas definirá el nuevo índice y redibujara la pantalla en base a este:

```
bottomNavigationBar: Container(
    height: 78,
    decoration: const BoxDecoration(
        border: Border(
            top: BorderSide(
                color: Color(0xFF0296E5),
                width: 1,
            ), // BorderSide
        ), // Border
    ), // BoxDecoration
    child: BottomNavigationBar(
        currentIndex: controller.index.value,
        onTap: (index) =>
            Get.find<BottomNavigatorController>().setIndex(index),
        backgroundColor: const Color(0xFF242A32),
        selectedItemColor: const Color(0xFF0296E5),
        unselectedItemColor: const Color(0xFF67686D),
        selectedFontSize: 12,
```

Por último, nos centraremos en crear cada una de los ítems de la barra de navegación, asociando cada una de ellas a una pantalla diferente, así como una asignación correspondiente de sus assets y el valor de su índice.

```
items: [
  BottomNavigationBarItem(
    icon: Container(
      margin: const EdgeInsets.only(bottom: 6),
      child: SvgPicture.asset(
        'assets/Home.svg',
        height: 21,
        width: 21,
        // ignore: deprecated_member_use
        color: controller.index.value == 0
          ? const Color(0xFF0296E5)
          : const Color(0xFF67686D),
      ), // SvgPicture.asset
    ), // Container
    label: 'Home',
  ), // BottomNavigationBarItem
```

```
BottomNavigationBarItem(
  icon: Container(
    margin: const EdgeInsets.only(bottom: 6),
    child: SvgPicture.asset(
      'assets/Search.svg',
      height: 21,
      width: 21,
      // ignore: deprecated_member_use
      color: controller.index.value == 1
        ? const Color(0xFF0296E5)
        : const Color(0xFF67686D),
    ), // SvgPicture.asset
  ), // Container
  label: 'Search',
  tooltip: 'Search Movies',
), // BottomNavigationBarItem
```

```
BottomNavigationBarItem(
  icon: Container(
    margin: const EdgeInsets.only(bottom: 6),
    child: SvgPicture.asset(
      'assets/Save.svg',
      height: 21,
      width: 21,
      // ignore: deprecated_member_use
      color: controller.index.value == 2
        ? const Color(0xFF0296E5)
        : const Color(0xFF67686D),
    ), // SvgPicture.asset
  ), // Container
  label: 'Watch list',
  tooltip: 'Your WatchList',
), // BottomNavigationBarItem
```

Cabe recalcar que, al iniciar nuestra aplicación, esta siempre comenzará por el ítem 0 de nuestra barra de navegación.

3.2.2 Home_Screen

Una de las pantallas mas cruciales durante la ejecución de nuestro programa. Esta permite al usuario ver un pantallazo de los actores mas populares o que son trending durante la semana, así como un pequeño ranking. Además, cuenta con un widget de búsqueda que le permite tener un workflow más satisfactorio y práctica.

Comenzamos declarando nuestra clase, heredándola a partir de la base StatelessWidget. Además, cargaremos nuestros controladores. Que serán usado por los demás widgets de nuestro programa:

- [PeopleController](#)
- [PeopleSearchController](#)

Los cargaremos usando GetX, mediante el método put():

```
class HomeScreen extends StatelessWidget {  
    HomeScreen({super.key});  
  
    final PeopleController pcontroller = Get.put(PeopleController());  
    final PeopleSearchController searchController =  
        Get.put(PeopleSearchController());
```

Comenzaremos sobrescribiendo la función build, a la cual le pasaremos como argumento nuestro BuildContext. Dentro de esta, crearemos un SingleChildScrollView para poder desplazarnos por el contenido de la aplicación de manera vertical:

```
@override  
Widget build(BuildContext context) {  
    return SingleChildScrollView(  
        child: Padding(  
            padding: const EdgeInsets.symmetric(  
                horizontal: 24,  
                vertical: 42,  
            ), // EdgeInsets.symmetric  
            child: Column(  
                mainAxisAlignment: MainAxisAlignment.start,  
                children: [
```

Dentro de esta, definiremos una columna, la cual contendrá un pequeño título y un [SearchBox](#). A este le pasaremos un VoidCallback, el cual, al confirmar nuestra búsqueda, pasará nuestra petición al motor. Además de esto, nos redirigirá a nuestra pantalla de [People Search Screen](#).

```
child: Column(  
    mainAxisAlignment: MainAxisAlignment.start,  
    children: [  
        const Align(  
            alignment: Alignment.centerLeft,  
            child: Text(  
                'What do you want to know?',  
                style: TextStyle(  
                    fontWeight: FontWeight.w600,  
                    fontSize: 24,  
                ), // TextStyle  
            ), // Text  
        ), // Align  
        const SizedBox(  
            height: 24,  
        ), // SizedBox  
        SearchBox(  
            onSumbit: () {  
                String search =  
                    Get.find<PeopleSearchController>().searchController.text;  
                Get.find<PeopleSearchController>().searchController.text = '';  
                Get.find<PeopleSearchController>().search(search);  
                Get.find<BottomNavigatorController>().setIndex(1);  
                FocusManager.instance.primaryFocus?.unfocus();  
            },  
        ), // SearchBox
```

Luego de esto, crearemos un objeto Obx que envolverá nuestro pequeño carrusel, para que este reaccione a los diferentes estados de carga de los ítems. Buscaremos el controlador iniciado anteriormente y establecemos su estado mediante un parámetro. Por último, crearemos un ListView, el cual se desplegará de manera horizontal y como ítems cargaremos nuestros Widgets [PeopleTopRatedItem](#).

```
Obx(
  () => pcontroller.isLoading.value
  ? const CircularProgressIndicator()
  : SizedBox(
    height: 300,
    child: ListView.separated(
      itemCount: pcontroller.mainTopRatedPeople.length,
      shrinkWrap: true,
      scrollDirection: Axis.horizontal,
      separatorBuilder: (_, __) => const SizedBox(width: 24),
      itemBuilder: (_, index) => PeopleTopRatedItem(
        person: pcontroller.mainTopRatedPeople[index],
        index: index + 1),
    ),
  ),
), // Obx
```

Crearemos un DefaultTabController para poder separar las diferentes categorías de actores a visualizar. Dentro de este, definiremos estilos y colores de este, así como los títulos de nuestros tabs:

```
DefaultTabController(
  length: 2,
  child: Column(
    mainAxisSize: MainAxisSize.min,
    children: [
      const TabBar(
        indicatorWeight: 3,
        indicatorColor: Color(
          0xFF3A3F47),
        labelStyle: TextStyle(fontSize: 11.0),
        tabs: [
          Tab(text: 'Populares'),
          Tab(text: 'Trending Weekly'),
        ],
      ),
    ],
  ),
)
```

Por último, definiremos el contenido de cada tab mediante el objeto [PeopleTabBuilder](#), al cual, le pasaremos una futura lista de Person, que, mediante el método de nuestra ApiService [getCustomPeople\(\)](#).

```
SizedBox(
  height: 400,
  child: TabBarView(children: [
    PeopleTabBuilder(
      future: ApiService.getCustomPeople(
        'person/popular?api_key=${Api.apiKey}&language=en-US&page=1'),
    ),
    PeopleTabBuilder(
      future: ApiService.getCustomPeople(
        'trending/person/week?api_key=${Api.apiKey}&language=en-US&page=1'),
    ),
  ]),
), // SizedBox
```

3.2.3 Details Screen

Esta pantalla es una de las principales responsables de mostrar información detallada de los actores y su relación con otras películas, además es el intermediario que nos permite guardar a nuestros actores favoritos para poder tenerlos siempre que queramos.

Empezaremos importando estas librerías y módulos:

```
import 'package:flutter/material.dart';
import 'package:flutter_svg/flutter_svg.dart';
import 'package:get/get.dart';
import 'package:movies/api/api.dart';

import 'package:movies/controllers/people_controller.dart';
import 'package:movies/models/person.dart';
```

Empezaremos definiendo nuestra clase heredándola de la base StatelessWidget. Además, definiremos nuestro constructor requiriendo un objeto Person, el cual mostrará los datos del actor:

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

Sobrescribiremos la función build, pasando como argumento un BuildContext. Luego, mediante GetX, el método find, buscaremos el controlador activo de [PeopleController](#) que nos ayudará a controlar y gestionar los flujos de datos. Ahora, empezaremos envolviendo todo dentro de un objeto Obx, el cual, permitirá al widget actuar de manera reactiva a las variables observables.

```
@override
Widget build(BuildContext context) {
  final PeopleController peopleController = Get.find<PeopleController>();
  return Scaffold(
    body: SafeArea(
      child: Obx(() {
        final p = peopleController.selectedPerson.value;
```

Ahora, Crearemos un SingleChildScrollView que nos servirá para poder ir desplazándonos por la toda la pantalla de manera vertical. A partir de aquí, empezaremos a construir todos los datos, comenzando por el encabezado. Dentro de un Row, crearemos un IconButton para para poder volver a la anterior pantalla (mediante el evento onPressed, usando el método Get.back()). Además, definiremos un texto “Details” y seguido de este un IconButton para guardar al actor en favoritos. Mediante el evento **onPressed** llamaremos al método PeopleController.[isInFavoriteList\(\)](#) al cual le pasaremos al actor seleccionado, que, dependiendo de su estado, cambiará la apariencia del ícono.

```
return SingleChildScrollView(
  child: Column(
    children: [
      // --- HEADER ---
      Padding(
        padding: const EdgeInsets.only(left: 24, right: 24, top: 34),
        child: Row(
          mainAxisAlignment: MainAxisAlignment.spaceBetween,
          crossAxisAlignment: CrossAxisAlignment.center,
          children: [
            IconButton(
              onPressed: () => Get.back(),
              icon: const Icon(Icons.arrow_back_ios,
                color: Colors.white), // Icon
            ), // IconButton
            const Text('Detail', style: TextStyle(fontSize: 24)),
            IconButton(
              onPressed: () => peopleController.addToFavoriteList(p),
              icon: Icon(
                peopleController.isInFavoriteList(p)
                  ? Icons.bookmark
                  : Icons.bookmark_outline,
                color: Colors.white,
                size: 33,
              ), // Icon
            ),
          ],
        ),
      ),
    ],
  ),
);
```

Ahora, crearemos un `SizedBox` donde cargaremos las imágenes de nuestro actor y las mostraremos en forma de banner. Además, mostraremos un pequeño recuadro con la imagen del actor.

```
// --- STACK DE IMÁGENES ---
SizedBox(
  height: 330,
  child: Stack(
    children: [
      ClipRRect(
        borderRadius: const BorderRadius.only(
          bottomLeft: Radius.circular(16),
          bottomRight: Radius.circular(16),
        ), // BorderRadius.only
        child: Image.network(
          Api.imageBaseUrl + p.profilePath,
          width: Get.width,
          height: 250,
          fit: BoxFit.cover,
        ), // Image.network
      ), // ClipRRect
      Container(
        margin: const EdgeInsets.only(left: 30),
        child: Align(
          alignment: Alignment.bottomLeft,
          child: ClipRRect(
            borderRadius: BorderRadius.circular(16),
            child: Image.network(
              'https://image.tmdb.org/t/p/w500/${p.profilePath}',
              width: 110,
              height: 140,
              fit: BoxFit.cover,
            ), // Image.network
          ), // ClipRRect
        ), // Align
      ), // Container
      Positioned(
        top: 255,
        left: 155,
        child: Text(
          p.name,
          style: const TextStyle(
            fontSize: 20, fontWeight: FontWeight.w500), // TextStyle
        ), // Text
      ), // Positioned
    ],
  ), // Stack
), // Container
```

Debajo de las imágenes, mostraremos algunas líneas de información del actor mediante el widget `_buildInfoBar`. Por último, generaremos unos tabs para ampliar la información y las relaciones de nuestro actor con `_buildTabs`, como su biografía.

```
const SizedBox(height: 25),
_buildInfoBar(p),
Material(
  color: Colors.transparent,
  child: _buildTabs(p),
), // Material
```

3.2.3.1 Build Info Bar

Este widget permite mostrar información relevante de un actor en concreto. Se requiere de un objeto `Person` como argumento, y se cargan assets personalizados que permiten volver más intuitiva la información mostrada dentro del Row.

```
Widget _buildInfoBar(Person p) {
  return Opacity(
    opacity: .6,
    child: Row(
      mainAxisAlignment: MainAxisAlignment.center,
      children: [
        SvgPicture.asset('assets/calender.svg'),
        const SizedBox(width: 5),
        Text(p.birthday ?? "..."),
        const Padding(
          padding: EdgeInsets.symmetric(horizontal: 10),
          child: Text('|')),
        SvgPicture.asset('assets/Ticket.svg'),
        const SizedBox(width: 5),
        Text(p.placeOfBirth ?? "..."),
      ],
    ),
  );
}
```

3.2.3.2 Build Tabs

Este widget construye los diferentes Tabs dentro de la pantalla de PersonDetailsScreen. Se requiere un objeto Person como argumento y dentro de este se definen la cantidad de tabs y los diferentes contenidos que se muestran en estos.

```
Widget _buildTabs(Person p) {
  return Padding(
    padding: const EdgeInsets.all(24),
    child: DefaultTabController(
      length: 3,
      child: Column(
        children: [
          const TabBar(
            indicatorColor: Color(0xFF3A3F47),
            tabs: [
              Tab(text: 'About Actor'),
              Tab(text: 'Movies/Series Participation'),
            ],
          ), // TabBar
          SizedBox(
            height: 400,
            child: TabBarView(
              children: [
                Padding(
                  padding: const EdgeInsets.only(top: 40),
                  child: Text(
                    p.biography ?? "Loading biography...",
                    textAlign: TextAlign.center,
                    style: const TextStyle(
                      fontSize: 16, fontWeight: FontWeight.w300),
                ), // Text
              ],
            ),
          ),
        ],
      ),
    ),
  );
}
```

3.2.4 People Search Screen

Esta pantalla permite ampliar los actores que se nos proporcionan en la home screen, permitiendo buscar personalizada mente cualquier actor que nosotros queramos.

Empezamos importando las siguientes librerías y módulos:

```
import 'package:fade_shimmer/fade_shimmer.dart';
import 'package:flutter/material.dart';
import 'package:flutter_svg/flutter_svg.dart';
import 'package:get/get.dart';
import 'package:movies/api/api.dart';
import 'package:movies/controllers/bottom_navigator_controller.dart';
import 'package:movies/controllers/people_search_controller.dart';
import 'package:movies/models/person.dart';
import 'package:movies/screens/person_details_screen.dart';
import 'package:movies/controllers/people_controller.dart';
import 'package:movies/widgets/person_infos.dart';
import 'package:movies/widgets/search_box.dart';
```

Ya que esta pantalla es más compleja y tendrá diferentes estados, definiremos nuestra clase heredando de StatelessWidget y sobrescribiremos la función createState y crearemos un nuevo estado.

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

  @override
  State<PeopleSearchScreen> createState() => _SearchScreenState();
}
```

Ahora, crearemos el nuevo estado, donde sobreescriviremos a la función build. Dentro de esta, usaremos GetX para buscar el controlador [PeopleController](#) que ya esté cargado.

Ahora, envolveremos todo dentro de un SingleChildScrollView para poder desplazarnos verticalmente por la pantalla. Dentro de este, tendremos definida dentro de una columna un row el cual contendrá un IconButton, que, al ser presionado, de disparará un evento que nos redirigirá a la pantalla de home. También, cargaremos un texto que nos muestre el título de la pantalla y crearemos un Tooltip donde mostraremos un mensaje que ayude al usuario a poder realizar acciones dentro de la interfaz.

```
class _SearchScreenState extends State<PeopleSearchScreen> {
  @override
  Widget build(BuildContext context) {
    final peopleController = Get.find<PeopleController>();

    return SingleChildScrollView(
      child: Padding(
        padding: const EdgeInsets.symmetric(horizontal: 24, vertical: 34),
        child: Column(
          children: [
            Row(
              mainAxisAlignment: MainAxisAlignment.spaceBetween,
              crossAxisAlignment: CrossAxisAlignment.center,
              children: [
                IconButton(
                  tooltip: 'Back to home',
                  onPressed: () =>
                      Get.find<BottomNavigatorController>().setIndex(0),
                  icon: const Icon(
                    Icons.arrow_back_ios,
                    color: Colors.white,
                  ), // Icon
                ),
                const Text(
                  'Search',
                  style: TextStyle(
                    fontWeight: FontWeight.w400,
                    fontSize: 24,
                  ), // TextStyle
                ), // Text
                const Tooltip(
                  message: 'Search your wanted actor here !',
                  triggerMode: TooltipTriggerMode.tap,
                  child: Icon(
                    Icons.info_outline,
                    color: Colors.white,
                  ), // Icon
                ), // Tooltip
              ],
            ),
          ],
        ),
      ),
    );
  }
}
```

Ahora, definimos un [SearchBox](#), al cual le pasaremos un VoidCallback, que, al confirmar nuestra búsqueda, pasará nuestra petición al motor. Además de esto, nos redirigirá a nuestra pantalla de [People Search Screen](#).

```
), // Row
const SizedBox(
  height: 40,
), // SizedBox
SearchBox(
  onSumbit: () {
    String search =
      Get.find<PeopleSearchController>().searchController.text;
    Get.find<PeopleSearchController>().searchController.text = '';
    Get.find<PeopleSearchController>().search(search);
    FocusManager.instance.primaryFocus?.unfocus();
  },
), // SearchBox
```

Crearemos un objeto Obx, el cual, envolverá los widgets hijos de forma reactiva, permitiéndoles redibujar o reaccionar a variables observables. Buscaremos y llamaremos a [PeopleSearchController](#) para cargar algunos estados.

```
Obx(
  () => Get.find<PeopleSearchController>().isLoading.value
    ? const CircularProgressIndicator()
    : Get.find<PeopleSearchController>().foundedPeople.isEmpty
```

En caso de que los resultados de la búsqueda sean nulos, se cargarán unos assets personalizados y un mensaje diciendo al usuario que no se ha encontrado ningun resultado, así como un pequeño tip.

```
? SizedBox(
  width: Get.width / 1.5,
  child: Column(
    children: [
      const SizedBox(
        height: 120,
      ), // SizedBox
      SvgPicture.asset(
        'assets/no.sv',
        height: 120,
        width: 120,
      ), // SvgPicture.asset
      const SizedBox(
        height: 10,
      ), // SizedBox
      const Text(
        'We Are Sorry, We Can Not Find The Actor :(',
        textAlign: TextAlign.center,
        style: TextStyle(
          fontSize: 28,
          fontWeight: FontWeight.w500,
          wordSpacing: 1,
        ), // TextStyle
      ), // Text
      const SizedBox(
        height: 10,
      ), // SizedBox
      const Opacity(
        opacity: .8,
        child: Text(
          'Find your actor by name, birthday etc ',
          textAlign: TextAlign.center,
          style: TextStyle(
            fontSize: 16,
            fontWeight: FontWeight.w200,
          ), // TextStyle
        ),
      ),
    ],
  ),
)
```

Crearemos un ListView para visualizar los resultados de las búsquedas. Mediante el controlador [PeopleSearchController](#) iremos viendo los diferentes estados de estos y dibujando las previsualizaciones. Además, crearemos un GestureDetector para que nuestro widget pueda reaccionar al gesto que deseemos hacerle, que en nuestro caso será “onTap”, redirigiéndonos a la pantalla [DetailsScreenPeople](#).

```

: ListView.separated(
  itemCount: Get.find<PeopleSearchController>()
    .foundedPeople
    .length,
  shrinkWrap: true,
  physics: const NeverScrollableScrollPhysics(),
  separatorBuilder: (_, __) =>
    const SizedBox(height: 24),
  itemBuilder: (_, index) {
    Person person = Get.find<PeopleSearchController>()
      .foundedPeople[index];
    return GestureDetector(
      onTap: () {
        peopleController.loadPersonDetails(person);
        // 2. Navegamos a la pantalla de detalles
        Get.to(() => DetailsScreenPerson());
      },
      child: Row(

```

Por último, definiremos el aspecto de nuestra previsualización, mostrando una imagen del actor encontrado, y a su costado, un breve información relacionada con este, que podremos generar gracias a la clase [InfosPerson](#).

```

child: Row(
  mainAxisAlignment: MainAxisAlignment.min,
  children: [
    ClipRRect(
      borderRadius: BorderRadius.circular(16),
      child: Image.network(
        Api.imageBaseUrl + person.profilePath,
        height: 180,
        width: 120,
        fit: BoxFit.cover,
        errorBuilder: (_, __, __) => const Icon(
          Icons.broken_image,
          size: 120,
        ), // Icon
        loadingBuilder: (_, __, __) {
          // ignore: no_wildcard_variable_uses
          if (__ == null) return __;
          return const FadeShimmer(
            width: 120,
            height: 180,
            highlightColor: Color(0xff22272f),
            baseColor: Color(0xff20252d),
          ); // FadeShimmer
        },
      ), // Image.network
    ), // ClipRRect
    const SizedBox(
      width: 20,
    ), // SizedBox
    InfosPerson(person: person)
  ],
)
```

3.2.5 Favorite List Screen

Esta pantalla nos permite visualizar los actores que tengamos guardados como favoritos.

Empezaremos importando las siguientes librerías y módulos:

```
import 'package:fade_shimmer/fade_shimmer.dart';
import 'package:flutter/material.dart';
import 'package:get/get.dart';
import 'package:movies/api/api.dart';
import 'package:movies/controllers/bottom_navigator_controller.dart';
import 'package:movies/controllers/people_controller.dart';
import 'package:movies/screens/person_details_screen.dart';
import 'package:movies/widgets/person_infos.dart';
```

Ahora definiremos nuestra clase heredando de la base StatelessWidget.

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

Por consiguiente, sobrescribiremos la función build, pasandole como argumento un BuildContext en el proceso.

Ahora, mediante GetX, buscaremos el [PeopleController](#) que ya tenemos definido por medio de la función Get.find.

```
@override
Widget build(BuildContext context) {
  final peopleController = Get.find<PeopleController>();
```

Envolveremos todo dentro de un objeto Obx, el cual reaccionará reactivamente con las personajes que vayamos guardando como favoritos.

Además, también crearemos un SingleChildScrollView para poder desplazarnos por la pantalla de ser necesarios.

```
return Obx(() => SingleChildScrollView(
  child: Padding(
    padding: const EdgeInsets.all(34.0),
    child: Column(
      children: [
```

Crearemos una columna y dentro de esta una Row, la cual contendrá un IconButton, que al ser presionado, nos redirigirá hacia la pantalla de HomeScreen de la aplicación mediante la función setIndex() del [BottomNavigatorController](#).

```
child: Column(
  children: [
    Row(
      mainAxisAlignment: MainAxisAlignment.spaceBetween,
      crossAxisAlignment: CrossAxisAlignment.center,
      children: [
        IconButton(
          tooltip: 'Back to home',
          onPressed: () =>
            | | Get.find<BottomNavigatorController>().setIndex(0),
          icon: const Icon(
            Icons.arrow_back_ios,
            color: Colors.white,
          ), // Icon
```

Seguido de esto, definiremos un título para presentar el tema de la pantalla:

```
const Text(
  'Favorites list',
  style: TextStyle(
    fontWeight: FontWeight.w400,
    fontSize: 24,
  ), // TextStyle
), // Text
const SizedBox(
  width: 33,
  height: 33,
), // SizedBox
```

En caso de encontrar a algun actor dentro de la lista de persons, este renderizará una previsualización por cada uno de los items dentro de la lista. Ademas, crearemos un objeto GestureDetector, el cual permitira al presionar sobre el widget llamar a un evento que nos refiriga a la pantalla de [DetailsScreen](#), no sin antes buscar y ampliar la información del actor mediante el método [loadPersonalDetails](#).

```
const SizedBox(
  height: 30,
), // SizedBox
if (Get.find<PeopleController>().favoriteListPeople.isNotEmpty)
...Get.find<PeopleController>().favoriteListPeople.map(
  (person) => Column(
    children: [
      GestureDetector(
        onTap: () {
          peopleController.loadPersonDetails(person);
          Get.to(() => DetailsScreenPerson());
        },
        child: Row(
          mainAxisAlignment: MainAxisAlignment.min,
          children: [
            ClipRRect(
              borderRadius: BorderRadius.circular(16),
              child: Image.network(
                Api.imageUrl + person.profilePath,
                height: 180,
                width: 100,
                fit: BoxFit.cover,
                errorBuilder: (_, __, ___) => const Icon(
                  Icons.broken_image,
                  size: 180,
                ), // Icon
              ),
            ),
          ],
        ),
      ),
    ],
  ),
)
```

Además, llamaremos al widget [InfosPerson](#), a la cual le pasaremos la persona seleccionada y será la encarga de cargar información al costado de la previsualización del actor.

```
  ],
  // Row
),
// SizedBox
InfosPerson(person: person)
],
```

En caso de que la lista esté vacía, simplemente crearemos un texto informativo, alertandonos de este estado.

```
if (Get.find<PeopleController>().favoriteListPeople.isEmpty)
const Column(
  children: [
    SizedBox(
      height: 200,
    ), // SizedBox
    Text(
      'No actors in your favorite list',
      style: TextStyle(
        fontSize: 20,
        fontWeight: FontWeight.w200,
      ), // TextStyle
    ),
  ],
), // Column
```

3.3 Controllers

Los controles son una parte fundamental del código para poder procesar los diferentes estados de nuestros objetos.

3.3.1 Bottom navigator controller

Este control funciona como parte fundamental de la navegación de nuestra aplicación. Permite envolver funcionalidades y widgets dentro de cada uno de sus índices y realizar procesos reactivos cada vez que nos desplazamos por estos.

Antes que nada, importaremos las siguientes librerías:

```
import 'package:get/get.dart';
import 'package:movies/screens/home_screen.dart';
import 'package:movies/screens/people_search_screen.dart';
import 'package:movies/screens/favorite_list_screen.dart';
```

Creamos una nueva instancia del controlador base y creamos una lista con todas las pantallas que tengamos disponibles:

- [Home Screen](#)
- [Details Screen](#)
- [Favorite List](#)

```
class BottomNavigatorController extends GetxController {
  var screens = [
    HomeScreen(),
    const PeopleSearchScreen(),
    const FavoriteList(),
  ];
```

definiremos PeopleSearchScreen y FavoriteList como constantes ya que estas no cambiarán durante el tiempo de ejecución del programa, de esta manera optimizando los procesos internos de estos.

Ademas, definiremos un indice como un integer observable, esto permitirá que GetX este observando esta variable por cambios que se lleguen a hacer en este. Además, definiremos las acciones que harán el evento setIndex, el cual al cambiar escoger una opción de entre todas las que hay en la barra de control, este cambiará su indice y por consiguiente asignará un valor nuevo a la varibla que previamente creamos.

```
var index = 0.obs;
void setIndex(indx) => index.value = indx;
```

3.3.2 People Controller

Este control es el encargado de gestionar los diferentes estados de los objetos que almacenan actores, por lo que usa de por medio GetX para la reactividad y permite tener los datos uniformes durante todo el tiempo de ejecución de nuestra aplicación.

Comenzamos importando las siguientes librerías y módulos:

```
import 'package:flutter/material.dart';
import 'package:get/get.dart';
import 'package:movies/api/api_service.dart';
import 'package:movies/models/person.dart';
```

Definimos la clase, que heredará de la base GetxController, y definiremos las variables globales de nuestra aplicación así como su estado. Es indispensable de que estas sean observables ".obs" puesto que necesitamos que estas puedan "escuchar" constantemente los cambios (creando un stream de promedio), y redibujando los widgets Obx que las estén usando.

```
//Controller for state management by the actors (People)
//Use Getx to drive reactivity on people list and actors details
class PeopleController extends GetxController {
    var isLoading = false.obs;
    var mainTopRatedPeople = <Person>[].obs;
    var favoriteListPeople = <Person>[].obs;

    var selectedPerson =
        Person(id: 0, name: '', profilePath: '', popularity: 0).obs;
```

Ahora, sobreescribiremos el método `onInit()`, el cual será llamado cuando se instancie nuestra clase. Este debe ser asíncrono, ya que necesitará hacer uso del método `getTopRatedActors()` de la clase ApiService para poder cargar un top de actores de forma automática al iniciar la aplicación.

```
@override
void onInit() async {
    isLoading.value = true;
    mainTopRatedPeople.value = (await ApiService.getTopRatedActors())!;
    isLoading.value = false;
    super.onInit();
}
```

A continuación, iremos explicando los diferentes métodos de nuestra clase y sus funcionalidades:

3.3.2.1 isInFavoriteList

Argumento: Objeto [Person](#).

Retorno: Boolean.

Esta función nos permite, mediante el método ".any" verificar si un actor está dentro de nuestra lista de favoritos:

```
//Check if the person is into favorite list people
bool isInFavoriteList(Person person) {
    return favoriteListPeople.any((p) => p.id == person.id);
```

3.3.2.2 addToListFavorite

Argumento: Objeto [Person](#).

Retorno: Void.

Este método nos permite agregar o remover a un actor dentro de nuestra lista de actores "favoriteListPeople". Llama al método `isInFavoriteList()` antes, y en caso de dar "True", este remueve de la lista al actor mediante el método `removeWhere()`. En caso de dar "False", añadirá al actor.

En ambos, se mostrará un Snackbar, con los detalles de la operación realizada:

```
void addToFavoriteList(Person person) {
    if (isInFavoriteList(person)) {
        favoriteListPeople.removeWhere((p) => p.id == person.id);
        Get.snackbar('Success', 'removed from favorite list',
            snackPosition: SnackPosition.BOTTOM,
            animationDuration: const Duration(milliseconds: 500),
            duration: const Duration(milliseconds: 500),
            backgroundColor: Color.fromARGB(255, 255, 255, 255));
    } else {
        favoriteListPeople.add(person);
        Get.snackbar('Success', 'added to favorite list',
            snackPosition: SnackPosition.BOTTOM,
            animationDuration: const Duration(milliseconds: 500),
            duration: const Duration(milliseconds: 500),
            backgroundColor: Color.fromARGB(255, 255, 255, 255));
    }
}
```

3.3.2.3 loadPersonDetails

Argumento: Objeto [Person](#).

Retorno: Future<Void>.

Definimos la función como asíncrona, puesto que esperamos una respuesta por parte de la función [getInfoPerson\(\)](#). Usado para poder cargar los datos de un actor, partiendo de la primicia de que este tiene información incompleta o muy básica.

```
Future<void> loadPersonDetails(Person person) async {
    isLoading.value = true;
    // Save basic data that we have to avoid overloading the screen with anything
    selectedPerson.value = person;
    //Call the API to complete the data
    var fullData = await ApiService.getInfoPeople(person.id);

    if (fullData != null) {
        selectedPerson.value = fullData;
    }
    isLoading.value = false;
}
```

3.3.3 People Search Controller

Este es el motor de búsqueda de nuestra aplicación para la sección de actores. Nos permite separar la lógica de negocio de la lógica de la interfaz.

Empezaremos importando las siguientes librerías y módulos:

```
import 'package:flutter/cupertino.dart';
import 'package:get/get.dart';
import 'package:movies/api/api_service.dart';
import 'package:movies/models/person.dart';
```

Heredamos nuestra clase de GetxController y comenzamos declarando un objeto de tipo searchController, este nos permitirá poder procesar el texto que deseamos usar para hacer la búsqueda.

Además declararemos algunas variables observables:

```
class SearchControllerPeople extends GetxController {
    TextEditingController searchController = TextEditingController();
    var searchText = ''.obs;
    var foundedPeople = <Person>[].obs;
    var isLoading = false.obs;
```

A continuación, hablaremos de los métodos de nuestra clase:

3.3.3.1 search

Argumento: String.

Retorno: Void.

Este método es el encargado procesar la búsqueda que deseamos hacer y almacenarla dentro de una lista de Personas “foundedPeople” mediante la función [getSearchedPeople\(\)](#).

```
void search(String query) async {
    isLoading.value = true;
    foundedPeople.value = (await ApiService.getSearchedPeople(query)) ?? [];
    isLoading.value = false;
}
```

3.4 Widgets

Los Widgets son lo que nos permite interactuar de manera dinámica con los servicios de la aplicación de manera visual.

3.4.1 Index Number

Este un widget que muestra los números que vemos en el carrusel haciendo alusión a la posición del ranking en la que están.

Comenzamos importando la siguiente librería:

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

Declararemos una clase que heredará de StatelessWidget. Seguido de esto, definiremos un constructor donde le pasaremos como argumento el número que deseemos mostrar:

```
class IndexNumber extends StatelessWidget {
  const IndexNumber({
    super.key,
    required this.number,
  });
  final int number;
```

Sobreescribiremos la función build, la cual nos pedirá el BuildContext y retornará un Widget. Pasando esto, veremos como este retornará un Texto el cual tendrá una serie de estilos integrados:

```
@override
Widget build(BuildContext context) {
  return Text(
    (number).toString(),
    style: const TextStyle(
      fontSize: 120,
      fontWeight: FontWeight.w600,
      shadows: [
        Shadow(
          offset: Offset(-1.5, -1.5),
          color: Color(0xFF0296E5),
        ), // Shadow
        Shadow(
          offset: Offset(1.5, -1.5),
          color: Color(0xFF0296E5),
        ), // Shadow
        Shadow(
          offset: Offset(1.5, 1.5),
          color: Color(0xFF0296E5),
        ), // Shadow
        Shadow(
          offset: Offset(-1.5, 1.5),
          color: Color(0xFF0296E5),
        ), // Shadow
      ],
      color: Color(0xFF242A32),
    ), // TextStyle
```

3.4.2 Person Infos

Este widget permite visualizar de manera ordenada datos adicionales del actor durante la previsualización de este.

Empezaremos importando las siguientes librerías y módulos:

```
import 'package:flutter/material.dart';
import 'package:flutter_svg/flutter_svg.dart';
import 'package:movies/models/person.dart';
```

Definiremos una clase heredandola de StatelessWidget, así como la definición de su constructor, requiriendo un objeto Person.

```
class InfosPerson extends StatelessWidget {
  const InfosPerson({super.key, required this.person});
  final Person person;
```

Sobreescribiremos el método build, el cual requerirá del BuildContext y retornará un SizedBox.

Dentro de este, almacenaremos una columna que mostrará el nombre del actor:

```
child: Column(
  mainAxisAlignment: MainAxisAlignment.start,
  mainAxisSize: MainAxisSize.spaceAround,
  children: [
    SizedBox(
      width: 200,
      child: Text(
        person.name,
        style: const TextStyle(
          fontSize: 20,
          fontWeight: FontWeight.w400,
          overflow: TextOverflow.ellipsis,
        ), // TextStyle
      ), // Text
    ), // SizedBox
```

Así como otra columna, la cual contendrá dos filas con el nivel de popularidad, lugar de nacimiento y fecha de cumpleaños. En cada uno de estos, se cargarán assets que bañan a juego con la información que se desea mostrar.

```
Column(
  mainAxisAlignment: MainAxisAlignment.start,
  children: [
    Row(
      children: [
        SvgPicture.asset('assets/Star.svg'),
        const SizedBox(
          width: 5,
        ), // SizedBox
        Text(
          person.popularity == 0.0
            ? 'N/A'
            : person.popularity.toString(),
          style: const TextStyle(
            fontSize: 16,
            fontWeight: FontWeight.w200,
            color: Color(0xFFFF8700),
          ), // TextStyle
        ), // Text
      ],
    ), // Row
  ],
)
```

```
Row(
  children: [
    SvgPicture.asset('assets/Ticket.svg'),
    const SizedBox(
      width: 5,
    ), // SizedBox
    Text(
      person.placeOfBirth ?? "Unknown",
      overflow: TextOverflow.ellipsis,
      maxLines: 1,
      style: const TextStyle(
        fontSize: 12,
        fontWeight: FontWeight.w200,
      ), // TextStyle
    ), // Text
  ],
), // Row
```

```
Row(
  children: [
    SvgPicture.asset('assets/calender.svg'),
    const SizedBox(
      width: 5,
    ), // SizedBox
    Text(
      person.birthday ?? "Unknown",
      overflow: TextOverflow.ellipsis,
      maxLines: 1,
      style: const TextStyle(
        fontSize: 16,
        fontWeight: FontWeight.w200,
      ), // TextStyle
    ), // Text
  ],
), // Row
```

3.4.3 People tab builder

Este widget es el encargado de construir las previsualizaciones de los actores. Es fundamental porque maneja la transición entre el estado de carga y visualización final, además de orquestar la navegación hacia los detalles.

Empezaremos importando las siguientes librerías y módulos:

```
import 'package:fade_shimmer/fade_shimmer.dart';
import 'package:flutter/material.dart';
import 'package:get/get.dart';
import 'package:movies/models/person.dart';
import 'package:movies/screens/person_details_screen.dart';
import 'package:movies/controllers/people_controller.dart';
```

Ahora definiremos la clase, la cual heredará de un StatelessWidget. Definiremos el constructor, el cual requerirá de una Future<List<Person>>.

```
class TabBuilderPeople extends StatelessWidget {
  const TabBuilderPeople({
    required this.future,
    super.key,
  });
  final Future<List<Person>> future;
```

Ahora sobrescribiremos el método build, el cual nos pedirá el BuildContext. Además, Buscaremos el controlador que en ese momento este activo de tipo [PeopleController](#).

```
@override
Widget build(BuildContext context) {
  // Search the controller to manage the navigation details.
  final peopleController = Get.find<PeopleController>();
```

Nos retornaran un Padding con los elementos. A su vez dentro de este estará un GridView, el cual construirá 3 columnas de actores con máximo de 6 items:

```
return Padding(
  padding: const EdgeInsets.only(top: 12.0, left: 12.0),
  child: FutureBuilder<List<Person>>(
    future: future,
    builder: (context, snapshot) {
      if (snapshot.hasData) {
        return GridView.builder(
          physics: const NeverScrollableScrollPhysics(),
          gridDelegate: const SliverGridDelegateWithFixedCrossAxisCount(
            crossAxisCount: 3,
            crossAxisSpacing: 15.0,
            mainAxisSpacing: 15.0,
            childAspectRatio: 0.6,
          ), // SliverGridDelegateWithFixedCrossAxisCount
          //The limite is 6 items
          itemCount: snapshot.data!.length > 6 ? 6 : snapshot.data!.length,
          itemBuilder: (context, index) {
            final personItem = snapshot.data![index];
          }
        );
      }
    }
);
```

Dentro de este, tendremos un GestureDetector, el cual permitirá percibir los gestos que realicemos sobre el objeto. Para la acción onTap(), el cual, llamará al método [loadPersonDetails\(\)](#) permitiendo actualizar nuestro actor con más información de la básica, para luego redireccionarnos a la pantalla de [DetailsScreenPerson\(\)](#):

```
final personItem = snapshot.data![index];

return GestureDetector(
  onTap: () {
    // 1. Loading extra data during the navigation
    peopleController.loadPersonDetails(personItem);
    // 2. Navigation to Details Screen
    Get.to(() => DetailsScreenPerson(person: personItem));
  },
);
```

Además, agregaremos un ClipRRect widget que nos permitirá darle una apariencia mas redondeada a las imágenes que deseemos usar como previsualización para nuestros actores:

```
child: ClipRRect(  
  borderRadius: BorderRadius.circular(16),  
  child: Image.network(  
    'https://image.tmdb.org/t/p/w500/${personItem.profilePath}',  
    height: 300,  
    width: 180,  
    fit: BoxFit.cover,  
    errorBuilder: (_, __, ___) => const Icon(  
      Icons.broken_image,  
      size: 50,  
      color: Colors.grey,  
, // Icon  
loadingBuilder: (_, child, loadingProgress) {  
  if (loadingProgress == null) return child;  
  return const FadeShimmer(  
    width: 180,  
    height: 250,  
    highlightColor: Color(0xff22272f),  
    baseColor: Color(0xff20252d),  
>; // FadeShimmer
```

3.4.4 People top rated item

Este widget permite visualizar a los actores con mayor ranking de popularidad, combinando el uso de Stacks para poder superponer el número de la posición ([Index Number](#)).

Empezaremos importando las siguientes librerías y módulos:

```
import 'package:fade_shimmer/fade_shimmer.dart';
import 'package:flutter/material.dart';
import 'package:get/get.dart';
import 'package:movies/api/api.dart';

import 'package:movies/models/person.dart';
import 'package:movies/controllers/people_controller.dart';
import 'package:movies/screens/person_details_screen.dart';
import 'package:movies/widgets/index_number.dart';
```

Creamos nuestra clase heredando de la base StatelessWidget y definiremos nuestro constructor, el cual requerirá de un Person y un int.

```
class PeopleTopRatedItem extends StatelessWidget {
  const PeopleTopRatedItem({
    super.key,
    required this.person,
    required this.index,
  });

  final Person person;
  final int index;
```

Sobreescribiremos la función build, encargada de retornar el Widget, además de pasarle como argumento el BuildContext. Mediante el metodo Get.find buscaremos el [PeopleController](#) que esté activo en ese momento y lo dejaremos definido. Además Envolveremos todo en un Stack, para poder superponer widgets.

Declararemos un GestureDetector para permitir definir acciones tras realizar un acción, que en nuestro caso, al presionar sobre el widget, mediante el controlador ubicado llamaremos a su metodo [loadPersonDetails\(\)](#) para cargar los datos extras del actor, y luego nos redirigirá a la pantalla [DetailsScreenPerson\(\)](#).

```
@override
Widget build(BuildContext context) {
  final controller = Get.find<PeopleController>();
  return Stack(
    children: [
      GestureDetector(
        onTap: () {
          controller.loadPersonDetails(person); // Triggers extra data upload
          Get.to(() => DetailsScreenPerson(person: person));
        },
    ],
}
```

Usaremos un ClipRRect para poder border nuestro contenedor, el cual contendrá la foto de nuestro actor. Además, definiremos aspectos de alineación y efectos de brillo.

```
child: Container(
  margin: const EdgeInsets.only(left: 12),
  child: ClipRRect(
    borderRadius: BorderRadius.circular(16),
    child: Image.network(
      Api.imageBaseUrl + person.profilePath,
      fit: BoxFit.cover,
      height: 250,
      width: 180,
      errorBuilder: (_, __, ___) => const Icon(
        Icons.broken_image,
        size: 180,
      ), // Icon
      loadingBuilder: (_, __, ___) {
        // ignore: no_wildcard_variable_use
        if (__ == null) return ___;
        return const FadeShimmer(
          width: 180,
          height: 250,
          highlightColor: Colors.color(0xff22272f),
          baseColor: Colors.color(0xff20252d),
        ); // FadeShimmer
      },
    ), // Image.network
```

Y el toque final, será llamar al Widget [Index Number](#), el cual estará superpuesto al widget base para poder darle una estética mas moderna.

```
// ...
Align(
  alignment: Alignment.bottomLeft,
  child: IndexNumber(number: index),
) // Align
```

3.4.5 Search Box

Este es la puerta de enlace que nos permitirá interactuar con nuestro motor de búsqueda.

Para empezar, importaremos las siguientes librerías y módulos:

```
import 'package:flutter/material.dart';
import 'package:flutter_svg/flutter_svg.dart';
import 'package:get/get.dart';
import 'package:movies/controllers/people_search_controller.dart';
```

Declararemos nuestra clase, heredandola de la base StatelessWidget. Además, definiremos su constructor el cual pedira un objeto VoidCallback como parámetro.

Un objeto VoidCallback, el cual es una función que no recibe y entrega nada y es usado ampliamente por widgets que esperan algun tipo de entrada.

```
class SearchBox extends StatelessWidget {
  const SearchBox({
    required this.onSubmit,
    super.key,
  });
  final VoidCallback onSubmit;
```

Ahora sobreescribiremos el metodo build, pasandole como argumento el BuildContext.

Dentro de este, retornaremos un TextField, el cual, dentro del parametro controller, usaremos GetX para buscar el control definido dentro del [PeopleSearchController](#).

```
@override
Widget build(BuildContext context) {
  return TextField(
    controller: Get.find<PeopleSearchController>().searchController,
```

Definiremos aspectos visuales básicos, así como cargar assets personalizados para mostrar como ícono de búsqueda.

```
style: const TextStyle(color: Colors.white),
decoration: InputDecoration(
  suffixIcon: IconButton(
    icon: Transform(
      alignment: Alignment.center,
      transform: Matrix4.rotationY(3.14),
      child: SvgPicture.asset(
        'assets/Search.svg',
        width: 22,
        height: 22,
      ),
    ), // SvgPicture.asset
  ), // Transform
  onPressed: () => onSubmit(),
```

Terminaremos de definir aspectos importantes, como bordeado, contorno, colores y hints:

```
border: OutlineInputBorder(
  borderRadius: BorderRadius.circular(16.0),
  borderSide: BorderSide.none,
), // OutlineInputBorder
hintStyle: const TextStyle(
  color: Color(
    0xFF67686D,
  ), // Color
  fontSize: 14,
  fontWeight: FontWeight.normal,
), // TextStyle
contentPadding: const EdgeInsets.only(
  left: 16,
  right: 0,
  top: 0,
  bottom: 0,
), // EdgeInsets.only
filled: true,
fillColor: const Color(0xFF3A3F47),
hintText: 'Search',
// Text
```

Por último, haremos uso otra vez de la función onSubmitted, pero esta a vez para lanzar nuestra búsqueda al presionar enter:

```
onSubmitted: (a) => onSubmitted(),
```

3.5 Models

El modelado de datos es una parte fundamental a la hora de desarrollar aplicación que tenga grupos de datos complejos. Es necesario crear clases especiales donde podamos crear la imagen de lo que queremos usar, para luego optimizar la forma en que lo usamos dentro de nuestro código, de manera más intuitiva.

Nuestro código base ya contaba con dos modelos hechos anteriormente:

```
import 'dart:convert';

class Movie {
    int id;
    String title;
    String posterPath;
    String backdropPath;
    String overview;
    String releaseDate;
    double voteAverage;
    List<int> genreIds;
    Movie({
        required this.id,
        required this.title,
        required this.posterPath,
        required this.backdropPath,
        required this.overview,
        required this.releaseDate,
        required this.voteAverage,
        required this.genreIds,
    });

    factory Movie.fromMap(Map<String, dynamic> map) {
        return Movie(
            id: map['id'] as int,
            title: map['title'] ?? '',
            posterPath: map['poster_path'] ?? '',
            backdropPath: map['backdrop_path'] ?? '',
            overview: map['overview'] ?? '',
            releaseDate: map['release_date'] ?? '',
            voteAverage: map['vote_average']?.toDouble() ?? 0.0,
            genreIds: List<int>.from(map['genre_ids']),
        ); // Movie
    }

    factory Movie.fromJson(String source) => Movie.fromMap(json.decode(source));
}

class Review {
    String author;
    String comment;
    double rating;
    Review({
        required this.author,
        required this.comment,
        required this.rating,
    });

    factory Review.fromJson(Map<String, dynamic> map) {
        return Review(
            author: map['name'] ?? '',
            comment: map['content'] ?? '',
            rating: map['rating']?.toDouble() ?? 0.0,
        );
    }
}
```

Pero en esta este proyecto, centraremos el desarrollo de nuestra aplicación, pensandolo para visualizar actores, por lo que crearemos un tercer modelo adicional para personas:

```
import 'dart:convert';

class Person {
    final int id;
    final String name;
    final String profilePath;
    final double popularity;
    final String? biography;
    final String? birthday;
    final String? placeOfBirth;
    Person({
        required this.id,
        required this.name,
        required this.profilePath,
        required this.popularity,
        this.biography,
        this.birthday,
        this.placeOfBirth,
    });

    factory Person.fromMap(Map<String, dynamic> map) {
        return Person(
            id: map['id'] as int,
            name: map['name'] ?? '',
            profilePath: map['profile_path'] ?? '',
            popularity: (map['popularity'] ?? 0.0).toDouble(),
            biography: map['biography'] ?? '',
            birthday: map['birthday'] ?? '',
            placeOfBirth: map['place_of_birth'] ?? '',
        );
    }

    factory Person.fromJson(String source) => Person.fromMap(jsonDecode(source));
}
```

Las factories fromMap y fromJson nos permitirán poder deserializar los archivos recibidos por el servidor mediante la API, sin importar que sean o diccionarios o JSONs

3.6 API

El módulo de API organiza y centraliza toda la lógica relacionada con la comunicación entre la aplicación y las distintas fuentes de datos externas. Su estructura permite mantener el código limpio, escalable y fácil de mantener, separando responsabilidades en archivos específicos que gestionan peticiones HTTP, modelos de datos, manejo de errores y utilidades comunes.

3.6.1 Api Credentials

Partiremos por lo esencial, las credenciales. A la hora de desarrollar aplicaciones con APIs es necesario hacer una investigación previa de los modos de uso que esta pueda tener. En nuestro caso, tendremos que registrarnos dentro de la web de TMBD API para poder obtener una credencial, y de esta manera, recibir una respuesta por parte del servidor cada vez que decidamos hacer peticiones.

En este módulo tendremos una clase que almacenará estos datos:

```
class Api {  
  static const baseUrl = "https://api.themoviedb.org/3/";  
  static const imageBaseUrl = "https://image.tmdb.org/t/p/original/";  
  static const apiKey = "9f9ac8ae339240fb5d6d73120a4b2827";  
}
```

3.6.2 Api Services

Ahora pasaremos a la parte mas importante de este módulo, el centro de servicios. Esta es la clase que permite realizar peticiones APIs a los servidores y luego transformarlas en objetos a partir de los [modelos](#) que tengamos. Iremos viendo los métodos ya predispuestos por el mismo código base y agregaremos algunos mas adicionales para adaptarlo a nuestro proyecto de actores.

Empezaremos importando los siguientes módulos y librerías:

```
import 'dart:convert';  
import 'package:movies/api/api.dart';  
import 'package:movies/models/movie.dart';  
import 'package:http/http.dart' as http;  
import 'package:movies/models/review.dart';  
import 'package:movies/models/person.dart';
```

Empezaremos analizando la estructura a detalle de uno de los métodos, puesto que tenderán a repetirse:

```
static Future<List<Person>?> getTopRatedActors() async {  
  List<Person> persons = [];  
  try {  
    http.Response response = await http.get(Uri.parse(  
      '${Api.baseUrl}person/popular?api_key=${Api.apiKey}&language=en-US&page=1'));  
    var res = jsonDecode(response.body);  
    res['results'].skip(6).take(5).forEach(  
      (p) => persons.add(  
        Person.fromMap(p),  
      ),  
    );  
    return persons;  
  } catch (e) {  
    return null;  
  }  
}
```

Partimos de la misma definición de este. La función `getTopRatedActors` retorna un objeto de tipo `Future<List?>`. En Dart, el tipo `Future` se utiliza para representar operaciones asíncronas, como llamadas a una API, que pueden tardar en completarse.

El `Future` actúa como una promesa: garantiza que en algún momento se devolverá un resultado, sin bloquear el hilo principal. El tipo dentro de los signos `< >` indica el valor que se espera recibir cuando la operación finalice. En este caso, se espera una lista de objetos `Person`, que puede ser `null` (por eso el signo `?` después de `List`).

Además, es muy importante agregar el keyword “async” para que el proceso sea asíncrono.

```
static Future<List<Person>> getTopRatedActors() async {
```

Inicializamos una variable del tipo de retorno de la función vacía y envolvemos el proceso de la petición dentro de un bloque try – catch (para un mayor control de excepciones).

Ahora, llamamos al módulo “http”, creando un objeto **Response** en el proceso, el cual será el responsable de manejar el envío y recepción de nuestra petición. Luego, definimos el objeto con la keyword **await**, el cual permite suspender la función hasta que se complete el Future, pero sin bloquear los demás procesos o tareas que se estén ejecutando, como componentes visuales de nuestra aplicación. Llamamos al método get del paquete http, al cual le pasaremos como argumento la URL del endpoint al que deseamos atacar, que en nuestro caso, iremos construyéndolo de manera más fácil haciendo uso de la clase [Api](#). Por último, definimos una variable de tipo var, la cual usaremos para almacenar la decodificación que hagamos con el método jsonDecode()

```
List<Person> persons = [];
try {
    http.Response response = await http.get(Uri.parse(
        '${Api.baseUrl}person/popular?api_key=${Api.apiKey}&language=en-US&page=1'));
    var res = jsonDecode(response.body);
```

De aquí en adelante, se irán presentando diversos casos, donde podremos recibir listas de objetos, creando un foreach para poder recogerlos, así como solo uno, llamando en todos los casos al constructor especial que hayamos definido:

```
res['results'].skip(6).take(5).forEach(
    (p) => persons.add(
        Person.fromMap(p),
    ),
);
return persons;
} catch (e) {
    return null;
}
```

```
    return Person.fromMap(res);
}
catch (e) {
    return null;
}
```

Visto todo esto, ya podremos crear cuantos métodos nos sean necesarios. Cabe recalcar que deberemos de adjuntar nuestra Apikey dentro de la URL para poder obtener una respuesta por parte del servidor:

A continuación, iremos explicando los diferentes métodos que usaremos para nuestra aplicación:

3.6.2.1 getTopRatedActors

Endpoint: <https://api.themoviedb.org/3/person/popular>

Argumento: None

Retorno: List<Person> || NULL

```
static Future<List<Person>> getTopRatedActors() async {
    List<Person> persons = [];
    try {
        http.Response response = await http.get(Uri.parse(
            '${Api.baseUrl}person/popular?api_key=${Api.apiKey}&language=en-US&page=1'));
        var res = jsonDecode(response.body);
        res['results'].skip(6).take(5).forEach(
            (p) => persons.add(
                Person.fromMap(p),
            ),
        );
    }
    return persons;
} catch (e) {
    return null;
}
```

3.6.2.2 getInfoPerson

Endpoint: [https://api.themoviedb.org/3/person/\\$VARIABLE](https://api.themoviedb.org/3/person/$VARIABLE)

Argumento: Int

Retorno: Person || NULL

```
static Future<Person?> getInfoPeople(int id) async {
    try {
        http.Response response = await http.get(Uri.parse(
            '${Api.baseUrl}person/$id?api_key=${Api.apiKey}&language=en-US'));
        var res = jsonDecode(response.body);
        return Person.fromMap(res);
    } catch (e) {
        return null;
    }
}
```

3.6.2.3 getCustomPeople

Endpoint: [https://api.themoviedb.org/3/person/\\$URL](https://api.themoviedb.org/3/person/$URL)

Argumento: String

Retorno: List<Person> || NULL

```
static Future<List<Person>?> getCustomPeople(String url) async {
    List<Person> people = [];
    try {
        http.Response response = await http.get(Uri.parse('${Api.baseUrl}$url'));
        var res = jsonDecode(response.body);
        res['results'].take(6).forEach(
            (m) => people.add(
                Person.fromMap(m),
            ),
        );
        return people;
    } catch (e) {
        return null;
    }
}
```

3.6.2.4 getSearchedPeople

Endpoint: <https://api.themoviedb.org/3/search/person>

Argumento: String

Retorno: List<Person> || NULL

```
static Future<List<Person>?> getSearchedPeople(String query) async {
    List<Person> people = [];
    try {
        http.Response response = await http.get(Uri.parse(
            'https://api.themoviedb.org/3/search/person?api_key=${Api.apiKey}&language=en-US&query=$query&page=1&include_adult=false'));
        var res = jsonDecode(response.body);
        res['results'].forEach(
            (p) => people.add(
                Person.fromMap(p),
            ),
        );
        return people;
    } catch (e) {
        return null;
    }
}
```

4. Ampliaciones

Una vez completada la aplicación principal, veremos de ampliar algunas de sus funcionalidades principales, permitiendo otorgar al usuario una mejor experiencia.

4.1 Ampliación 1

En esta ampliación nos centraremos en añadir en la página de detalle del actor, en la parte inferior, una lista con las películas en las que ha participado el actor seleccionado.

Si se pulsa sobre una película, se mostrará todos los datos relacionados con ella.

Antes de comenzar, será imprescindible que tengamos nuestro modelo de películas listo. En nuestro caso, el código base ya venía con todos los modelos hechos, por lo que solo los reutilizaremos([Models](#)).

Ahora, haremos modificaciones dentro del módulo API > [ApiServices](#), dentro de este agregaremos un nuevo método estático encargado de retornarnos una futura lista de películas, basada en las participaciones que haya tenido un actor en ellas. Usaremos el id del actor para identificarlo.

```
static Future<List<Movie>> getPersonMovieCredits(int personId) async {
  List<Movie> movies = [];
  try {
    http.Response response = await http.get(Uri.parse(
      '${Api.baseUrl}person/$personId/movie_credits?api_key=${Api.apiKey}&language=en-US'));
    var res = jsonDecode(response.body);

    // Endpoint returns a list called 'cast'
    res['cast'].forEach((m) {
      movies.add(Movie.fromMap(m));
    });
    return movies;
  } catch (e) {
    return null;
  }
}
```

Ahora, modificaremos el modulo Controllers > [People Controller](#), en el cual añadiremos una nueva variable observable, así como una modificación dentro del método loadPersonDetails(). Dentro de este, limpiaremos la variable cada vez que usemos el método, luego, haremos la llamada en paralelo de nuestros métodos Api para mejorar la eficiencia. Por último, guardaremos los resultados en sus respectivas variables.

```
var personMovies = <Movie>[].obs;

// loadPersonDetails Updated
Future<void> loadPersonDetails(Person person) async {
  isLoading.value = true;
  selectedPerson.value = person;
  personMovies.clear(); // Clear Actor's movies before

  // Parallel call to improve performance
  var results = await Future.wait([
    ApiService.getInfoPerson(person.id),
    ApiService.getPersonMovieCredits(person.id),
  ]);

  if (results[0] != null) selectedPerson.value = results[0] as Person;
  if (results[1] != null) personMovies.value = results[1] as List<Movie>;

  isLoading.value = false;
}
```

En este punto, tendremos que asegurarnos tener listo un controlador para las películas. En nuestro caso, el código base ya nos proporcionaba uno, por lo que lo reutilizaremos. Este controlador hereda de la base GetxController, y cuenta con las siguientes variables observables:

```
class MoviesController extends GetxController {
  var isLoading = false.obs;
  var mainTopRatedMovies = <Movie>[].obs;
  var watchListMovies = <Movie>[].obs;
```

Ademas, cuenta con una sobrescritura del método `onInit`, el cual hace uso de uno de los servicios de Api Service, el cual nos retorna una lista de las top peliculas actuales:

```
class ApiService {
    static Future<List<Movie>> getTopRatedMovies() async {
        List<Movie> movies = [];
        try {
            http.Response response = await http.get(Uri.parse(
                '${Api.baseUrl}movie/top_rated?api_key=${Api.apiKey}&language=en-US&page=1'));
            var res = jsonDecode(response.body);
            res['results'].skip(6).take(5).forEach(
                (m) => movies.add(
                    Movie.fromMap(m),
                ),
            );
        }
        return movies;
    } catch (e) {
        return null;
    }
}

@Override
void onInit() async {
    isLoading.value = true;
    mainTopRatedMovies.value = (await ApiService.getTopRatedMovies())!;
    isLoading.value = false;
    super.onInit();
}
```

Además, cuenta con métodos parecidos a los del `PeopleController`, los cuales resumiremos a continuación:

- `isInWatchList` es un método que devuelve un booleano cuando encuentra una película dentro de la lista `watchListMovies`.

```
bool isInWatchList(Movie movie) {
    return watchListMovies.any((m) => m.id == movie.id);
}
```

- `addToWatchList` es un método que añade una pelicula a la lista de peliculas para ver mas tarde, pero si esta ya se encuentra dentro, la remueve. Seguido de desto, nos muestra un Snack bar con la información de la acción realizada.

```
void addToWatchList(Movie movie) {
    if (isInWatchList(movie)) {
        watchListMovies.remove(movie);
        Get.snackbar('Success', 'removed from watch list',
            snackPosition: SnackPosition.BOTTOM,
            animationDuration: const Duration(milliseconds: 500),
            duration: const Duration(milliseconds: 500));
    } else {
        watchListMovies.add(movie);
        Get.snackbar('Success', 'added to watch list',
            snackPosition: SnackPosition.BOTTOM,
            animationDuration: const Duration(milliseconds: 500),
            duration: const Duration(milliseconds: 500));
    }
}
```

Una vez tengamos esto, será fundamental que declaremos este controlador en nuestro [Home](#):

```
final PeopleController pcontroller = Get.put(PeopleController());
final PeopleSearchController searchController =
    Get.put(PeopleSearchController());
final MoviesController mcontroller = Get.put(MoviesController());
```

Ahora, empezaremos a hacer modificaciones dentro de [Person Details Screen](#). Dentro de la función `Build Tabs`, encargada de construir el contenido de los diferentes tabs existentes, extenderemos uno de estos apartados. Aquí, envolveremos todos los demás objetos dentro de un objeto `Obx`, para que puedan responder reactivamente a cambios en las variables

observables. Además, nos aseguraremos de colocar un mensaje informativo en caso de que aún se esté cargando las películas, o no se hayan encontrado resultados:

```
Obx(() {
    // If still overloading movies list
    if (pc.isLoading.value) {
        return const Center(child: CircularProgressIndicator());
    }

    // if doesn't have any movie
    if (pc.personMovies.isEmpty) {
        return const Center(child: Text("No movies found"));
    }
})
```

Ahora, construiremos un GridView, donde lo parametrizaremos basandonos en los resultados de nuestra película. Además, crearemos un GestureDetector para que, al tocar algunos de los recuadros, este nos rediriga a la pantalla de Movies Details Screen, donde expandiremos mas los detalles de esta.

```
child: ClipRRect(
    borderRadius: BorderRadius.circular(12),
    child: Image.network(
        'https://image.tmdb.org/t/p/w500${movie.posterPath}',
        fit: BoxFit.cover,
        errorBuilder: (_, __, ___) => Container(
            color: Colors.white,
            child: const Icon(Icons.movie,
                color: Colors.white), // Icon
        ), // Container

    return GridView.builder(
        padding: const EdgeInsets.only(top: 20),
        gridDelegate:
            const SliverGridDelegateWithFixedCrossAxisCount(
                crossAxisCount: 3,
                mainAxisSpacing: 10,
                crossAxisSpacing: 10,
                childAspectRatio: 0.65,
            ), // SliverGridDelegateWithFixedCrossAxisCount
        itemCount: pc.personMovies.length,
        itemBuilder: (context, index) {
            final movie = pc.personMovies[index];
            return GestureDetector(
                onTap: () {
                    Get.to(() => MoviesDetailsScreen(movie: movie));
                },
            );
        },
    );
})
```

Ahora, veremos cómo está construida la pantalla de detalles de las películas. Hay que recalcar que es muy parecida a la pantalla de [Person Details Screen](#), por lo que solo mencionaremos algunos aspectos claves de la clase. Cuenta con una opción para guardar las películas, llamando a MovieController.

```
Tooltip(
    message: 'Save this movie to your watch list',
    triggerMode: TooltipTriggerMode.tap,
    child: IconButton(
        onPressed: () {
            Get.find<MoviesController>().addToWatchList(movie);
        },
        icon: Obx(
            () =>
                Get.find<MoviesController>().isInWatchList(movie)
                    ? const Icon(
                        Icons.bookmark,
                        color: Colors.white,
                        size: 33,
                    ) // Icon
                    : const Icon(
                        Icons.bookmark_outline,
                        color: Colors.white,
                        size: 33,
                    ), // Icon
        ), // Obx
    ),
)
```

Esta pantalla cuenta con dos tabs, una para mostrar la sinopsis, así como las reviews de esta, usando uno de los [modelos base](#) que ya venian con el código base.

```

    tabs: [
      Tab(text: 'About Movie'),
      Tab(text: 'Reviews'),
    ],
  ), // TabBar

```

De esta manera, mediante esta pantalla no solo somos capaces de poder observar detalles importantes de la película en sí, sino también las opiniones que tiene otras personas de esta:

```

FutureBuilder<List<Review>>(
  future: ApiService.getMovieReviews(movie.id),
  builder: (_, snapshot) {
    if (snapshot.hasData) {
      return snapshot.data!.isEmpty
        ? const Padding(
            padding: EdgeInsets.only(top: 30.0),
            child: Text(
              'No review',
              textAlign: TextAlign.center,
            ),
          )
        : ListView.builder(
            itemCount: snapshot.data!.length,
            itemBuilder: (_, index) => Padding(
              padding: const EdgeInsets.all(10.0),
              child: Row(
                crossAxisAlignment:
                  CrossAxisAlignment.start,
                children: [
                  Column(
                    children: [
                      SvgPicture.asset(
                        'assets/avatar.svg',
                        height: 50,
                        width: 50,
                        // fit: BoxFit.cover,
                      ),

```

```

                    Text(
                      snapshot.data![index].rating
                        .toString(),
                      style: const TextStyle(
                        color: Color(0xff0296E5),
                      ),
                    ),
                  ],
                ),
              ),
            ),
            const SizedBox(
              width: 10,
            ),
          ),
        ),
      ),
      Column(
        crossAxisAlignment:
          CrossAxisAlignment.start,
        children: [
          Text(
            snapshot.data![index].author,
            style: const TextStyle(
              fontSize: 16,
              fontWeight:
                FontWeight.w400,
            ),
          ),
          const SizedBox(
            height: 10,
          ),
          SizedBox(
            width: 245,
            child: Text(
              snapshot.data![index].comment,
              style: const TextStyle(
                fontSize: 8
              ),
            ),
          ),

```

De esta manera, tendremos nuestra aplicación funcionando correctamente, y lista para poder ser usada (para ver una demostración de esta, ir al [Demo](#))

4.2 Ampliación 2 y 3

En esta ampliación se nos ha pedido que, mediante un menú, podamos escoger entre ver películas o actores, y se cargue una interfaz relacionada con esta (Si se desea ver una demostrativa, ir al [enlace](#)).

Además, para la segunda ampliación, se nos ha pedido que usemos un menú “búrguer” para poder escoger entre películas o series, y luego mostrar la información pertinente a esta.

He considerado de combinar ambas ampliaciones en una sola, permitiendo al usuario mediante un solo menú “burguer” poder escoger entre ver películas, series o actores.

Empezaremos creando un control maestro heredado de la clase base GetxController, el cual, nos permitirá controlar los diferentes estados generales de la aplicación cuando naveguemos por el menú. Podremos establecer los diferentes estados con el enum AppModule, y siempre empezaremos por defecto por el módulo de Actores. Además, cuando cambies de módulo, este nos redirigirá siempre a la pantalla de Home del módulo.

```
< import 'package:movies/controllers/bottom_navigator_controller.dart';
< import 'package:get/get.dart';

enum AppModule { movies, actors, series }

< class AppModuleController extends GetxController {
    // Start with "Actors"
    var selectedModule = AppModule.actors.obs;

    void changeModule(AppModule module) {
        selectedModule.value = module;
        // Always when category changes, the Navigator index will be reset to home screen.
        Get.find<BottomNavigatorController>().setIndex(0);
    }
}
```

Aquí haremos una modificación importante al control [Bottom Navigator Controller](#), y es que ahora este contendrá una lista de widgets, los cuales serán el conjunto de las diferentes pantallas de los diferentes módulos, por lo que, al cambiar de módulo mediante el menú, podremos cambiar la distribución de los widgets:

```
class BottomNavigatorController extends GetxController {
    var index = 0.obs;

    // Screens for the actor module.
    List<Widget> actorScreens = [
        PeopleHomeScreen(),
        PeopleSearchScreen(),
        FavoriteList(),
    ];

    // Screens for the movies module.
    List<Widget> movieScreens = [
        MoviesHomeScreen(),
        MoviesSearchScreen(),
        MoviesWatchList(),
    ];
}
```

Por último, para poder conectar todos nuestros módulos de manera óptima, buscaremos unificarlos todos con la creación de un menú búrguer, por lo que deberemos hacer cambios importantes dentro del [Main](#). A continuación, iremos analizando paso a paso los cambios realizados dentro de la clase y veremos como afectan al funcionamiento de la aplicación.

El widget Main utiliza el componente **Scaffold** para definir el armazón de la interfaz, integrando un **AppBar** dinámico cuyo título cambia reactivamente según el módulo seleccionado por el usuario. La pieza central de la navegación es el **IndexedStack**, que permite mantener el estado de las páginas (como el scroll o los datos cargados) al cambiar entre las pestañas de la barra inferior. Este componente es fundamental para la optimización

del rendimiento, ya que evita la reconstrucción constante de las vistas, intercambiando simplemente el set de pantallas que devuelve la lógica de módulos.

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

  final BottomNavigatorController navLinesController =
    Get.put(BottomNavigatorController());
  final AppModuleController moduleController = Get.put(AppModuleController());

  @override
  Widget build(BuildContext context) {
    return Obx(
      () => GestureDetector(
        onTap: () => FocusScope.of(context).unfocus(),
        child: Scaffold(
          drawer: _buildDrawer(moduleController),
          appBar: AppBar(
            elevation: 0,
            backgroundColor: const Color(0xFF242A32),
            title: Text(
              moduleController.selectedModule.value == AppModule.actors
                ? "Actors"
                : "Movies",
              style: const TextStyle(
                fontWeight: FontWeight.w600,
                color: Color.fromARGB(255, 255, 255, 255)), // TextStyle
            ), // Text
        ), // AppBar
    ), // Scaffold
  );
}
```

La implementación se apoya fuertemente en **GetX** para la gestión del estado y las dependencias. Mediante `Get.put()`, se aseguran las instancias de los controladores de navegación y de módulos desde el inicio del ciclo de vida del widget. El uso del widget **Obx** permite que toda la interfaz sea "sensible" a los cambios en tiempo real: cuando un usuario selecciona una nueva categoría en el menú burger, el controlador notifica el cambio, provocando que tanto el cuerpo de la aplicación como los elementos visuales del AppBar y el BottomBar se actualicen instantáneamente sin necesidad de recargar manualmente la pantalla.

```
Widget _buildDrawer(AppModuleController controller) {
  return Drawer(
    backgroundColor: const Color(0xFF242A32),
    child: ListView(
      padding: EdgeInsets.zero,
      children: [
        const DrawerHeader(
          decoration: BoxDecoration(color: Color(0xFF0296E5)),
          child: Text("TMDB ", style: TextStyle(color: Colors.white, fontSize: 24)), // Text
        ), // DrawerHeader
        ListTile(
          leading: const Icon(Icons.person, color: Colors.white),
          title: const Text("Actors", style: TextStyle(color: Colors.white)),
          onTap: () => controller.changeModule(AppModule.actors),
        ), // ListTile
        ListTile(
          leading: const Icon(Icons.movie, color: Colors.white),
          title: const Text("Movies", style: TextStyle(color: Colors.white)),
          onTap: () => controller.changeModule(AppModule.movies),
        ), // ListTile
      ],
    ), // ListView
  );
}
```

Para mantener un código limpio y legible, la interfaz se ha descompuesto en métodos especializados como `_buildDrawer`, `_buildBottomBar` y `_navItem`. El **Drawer** funciona como el selector de contexto de alto nivel, permitiendo al usuario saltar entre el ecosistema de Actores y el de Películas. Por otro lado, la **BottomNavigationBar** gestiona la navegación interna de cada módulo (Inicio, Búsqueda y Favoritos), utilizando activos SVG personalizados para mantener una estética moderna y profesional. Esta separación de responsabilidades facilita enormemente la escalabilidad, permitiendo añadir el módulo de Series simplemente registrando una nueva lista de pantallas.

Recursos

Demos

Demo TMDB App principal (Actors)

Enlace:

- [Demo_1](#)

Demo TMDB App Ampliación 1

Enlace:

- [Demo_2](#)

Demo TMDB App Ampliación 2 y 3

Enlace:

- [Demo_3](#)

Librerías

Get

- [get | Flutter package](#)

flutter_svg

- [flutter_svg | Flutter package](#)

http

- [http | Dart package](#)

fade_shimmer

- [fade_shimmer | Flutter package](#)

flutter_launcher_icons

- [flutter_launcher_icons | Dart package](#)

APIs

TMDB API

- [Getting Started](#)

Repositorios GitHub

Repositorio Personal:

- [Mobile_projects_2526](#)

Repositorio Proyecto base TheMovieDb:

- [sergicarreras/movies2526](#)