DOCENTE: JOSE ARTURO BUSTAMANTE LAZCANO

ALUMNA:WENDOLINE ELIZABETH HERNANDEZ PONCE

Portafolio de Evidencias agosto – diciembre 2025

Desarrollo de aplicaciones moviles

Índice

[Semana 1: Introducción y configuración del entorno 3](#_Toc216358159)

[**Introducción y Configuración del Entorno** 5](#_Toc216358160)

[EJEMPLO: 6](#_Toc216358161)

[Semana 2: Fundamentos de Dart 7](#_Toc216358162)

[**Fundamentos de Dart** 9](#_Toc216358163)

[EJEMPLO: 10](#_Toc216358164)

[Semana 3: Widgets básicos de Flutter 11](#_Toc216358165)

[**Widgets Básicos de Flutter** 13](#_Toc216358166)

[EJEMPLO: 14](#_Toc216358167)

[Semana 4: Manejo de imágenes e íconos 15](#_Toc216358168)

[**Manejo de Imágenes e Íconos** 17](#_Toc216358169)

[EJEMPLO: 17](#_Toc216358170)

[Semana 5: Navegación entre pantallas 19](#_Toc216358171)

[**Navegación entre Pantallas** 20](#_Toc216358172)

[EJEMPLO: 21](#_Toc216358173)

[Semana 6: Stateful Widgets y manejo del estado 24](#_Toc216358174)

[**Stateful Widgets y Manejo del Estado** 25](#_Toc216358175)

[EJEMPLO: 26](#_Toc216358176)

[Semana 7: Formularios e inputs 28](#_Toc216358177)

[**Formularios e Inputs** 29](#_Toc216358178)

[EJEMPLO: 30](#_Toc216358179)

[Semana 8: Listas y manejo de scroll 32](#_Toc216358180)

[**Listas y Manejo de Scroll** 33](#_Toc216358181)

[EJEMPLO: 34](#_Toc216358182)

[Semana 9: Consumo de APIs (Parte 1: HTTP y JSON) 35](#_Toc216358183)

[**Consumo de APIs (Parte 1: HTTP y JSON)** 36](#_Toc216358184)

[EJEMPLO: 37](#_Toc216358185)

[Semana 10: Consumo de APIs (Parte 2: FutureBuilder y listas) 38](#_Toc216358186)

[**Consumo de APIs (Parte 2: FutureBuilder y Listas)** 40](#_Toc216358187)

[EJEMPLO: 40](#_Toc216358188)

[Semana 11: Persistencia local con SharedPreferences 42](#_Toc216358189)

[**Persistencia local con SharedPreferences** 43](#_Toc216358190)

[EJEMPLO: 44](#_Toc216358191)

[Semana 12: Bases de datos locales con SQLite 47](#_Toc216358192)

[**Bases de datos locales con SQLite** 48](#_Toc216358193)

[EJEMPLO: 49](#_Toc216358194)

[Semana 13: Diseño avanzado y temas 51](#_Toc216358195)

[**Diseño avanzado y temas** 52](#_Toc216358196)

[EJEMPLO: 53](#_Toc216358197)

[Semana 14: Animaciones en Flutter 55](#_Toc216358198)

[**Animaciones en Flutter** 57](#_Toc216358199)

[EJEMPLO: 57](#_Toc216358200)

[Semana 15: Integración con Firebase – Autenticación 59](#_Toc216358201)

[**Integración con Firebase – Autenticación** 61](#_Toc216358202)

[EJEMPLO: 61](#_Toc216358203)

[Semana 16: Integración con Firebase – Firestore 64](#_Toc216358204)

[**Integración con Firebase – Firestore** 67](#_Toc216358205)

[EJEMPLO: 68](#_Toc216358206)

# Semana 1: Introducción y configuración del entorno

**Introducción al desarrollo con Flutter**

Durante la semana 1 se presenta el fundamento del desarrollo de aplicaciones utilizando Flutter. Se explica que Flutter es un framework creado por Google que permite crear aplicaciones para Android, iOS, web y escritorio utilizando un solo código base. También se aclara que utiliza el lenguaje de programación Dart, un lenguaje moderno, eficiente y especialmente diseñado para interfaces visuales de alta velocidad. En esta etapa el estudiante comprende las ventajas de trabajar con un framework multiplataforma y cómo este enfoque facilita el desarrollo profesional de aplicaciones.

**Instalación y preparación del entorno de trabajo**

En esta semana se dedica tiempo a la instalación y configuración completa del entorno de desarrollo. El estudiante descarga el SDK de Flutter desde la página oficial y lo configura dentro del sistema operativo para poder ejecutar comandos desde la terminal. Se revisa el uso del comando flutter doctor, que sirve para verificar si todo el sistema está correctamente instalado, incluyendo el SDK de Android, las herramientas de compilación y la conexión con dispositivos. También se instala Android Studio o Visual Studio Code, y se añaden las extensiones necesarias de Flutter y Dart para comenzar a programar correctamente.

**Creación de emuladores y pruebas iniciales**

Se enseña a configurar un emulador dentro de Android Studio, utilizando el Administrador de Dispositivos Virtuales para simular un teléfono Android. Este emulador permite probar la aplicación sin necesidad de un dispositivo físico. Se explican los requisitos mínimos que debe tener un emulador y cómo encenderlo para usarlo durante la programación. En caso de utilizar un celular real, se explica cómo activar las opciones de desarrollador y el modo de depuración USB.

**Creación del primer proyecto Flutter**

Una actividad fundamental de la semana consiste en crear el primer proyecto utilizando el comando flutter create. Al abrir el proyecto, el estudiante analiza la estructura predeterminada que genera Flutter. Se estudia la carpeta lib, que contiene el archivo principal en Dart; la carpeta android y la carpeta ios, que contienen configuraciones específicas de cada plataforma; y el archivo pubspec.yaml, que sirve para importar dependencias, imágenes, fuentes y otros elementos importantes. Se observa cómo Flutter organiza automáticamente estos componentes.

**Comprensión del archivo main.dart**

El estudiante revisa el archivo main.dart, que contiene la función main, encargada de iniciar la aplicación. Se estudia el uso del método runApp y la estructura de widgets iniciales, como MaterialApp, Scaffold y Text. También se explica la filosofía de Flutter basada en widgets, donde todo en la interfaz es un widget, desde un simple texto hasta estructuras completas de diseño.

**Ejecución de la aplicación y validación del entorno**

Finalmente, se ejecuta la primera aplicación utilizando flutter run. El estudiante visualiza la aplicación en el emulador y confirma que el entorno de Flutter funciona correctamente. Esta ejecución inicial permite observar el uso de Hot Reload, una herramienta que actualiza los cambios del código en tiempo real sin detener la aplicación, lo que facilita enormemente el desarrollo. Con esta actividad se completa la preparación del entorno necesario para avanzar a temas más complejos en las siguientes semanas.

| **Tema Principal** | **Conceptos Clave** | **Herramientas y Comandos** |
| --- | --- | --- |
| **Introducción a Flutter/Dart** | Flutter: Framework de UI, Creado por Google. Dart: Lenguaje de programación optimizado para interfaces y Hot Reload. **Ventajas:** Código base único (Multiplataforma), Alto rendimiento (AOT/JIT). | SDK de Flutter, Dart, Android/iOS SDK, Emulador AVD, flutter doctor, flutter create. |
| **Estructura del Proyecto** | **lib:** Contiene el código fuente (main.dart). **pubspec.yaml:** Gestión de dependencias (paquetes) y recursos (assets). **android/ios:** Configuración nativa específica. | **main.dart**, pubspec.yaml, Carpetas lib, assets. |
| **Ejecución y Debug** | **Hot Reload:** Recarga los cambios de UI en milisegundos sin perder el estado de la aplicación. **runApp():** Punto de entrada que renderiza el widget raíz. | flutter run, Hot Reload, runApp(). |

**Introducción y Configuración del Entorno**

**Foco Adicional:** **La Ventaja de Dart y la Compilación.**

* **Dart JIT vs. AOT:** Se resalta que Dart utiliza **Just-in-Time (JIT)** compilation en el desarrollo (lo que permite el **Hot Reload** rápido) y **Ahead-of-Time (AOT)** compilation para producción, compilando a código nativo de ARM, lo que garantiza el alto rendimiento de las aplicaciones Flutter.
* **Editor Setup:** La importancia de las extensiones de Flutter/Dart radica en la autocompletación de código, el formateo automático (Ctrl+Shift+I o Cmd+Shift+I) y la función **Code-Folding** para manejar el anidamiento profundo de widgets.
* **Estructura:** El archivo pubspec.lock es tan importante como pubspec.yaml, ya que asegura que todos los desarrolladores usen exactamente la misma versión de cada dependencia.

# EJEMPLO:

import 'package:flutter/material.dart';

void main() {

// Inicia la aplicación ejecutando el widget raíz

runApp(const MiPrimeraApp());

}

// Widget raíz, inmutable, define el tema principal

class MiPrimeraApp extends StatelessWidget {

const MiPrimeraApp({super.key});

@override

Widget build(BuildContext context) {

return MaterialApp(

title: 'Flutter Demo',

// Widget principal que estructura la pantalla

home: Scaffold(

appBar: AppBar(

title: const Text('Semana 1: Configuración'),

backgroundColor: Colors.blue,

),

body: const Center(

child: Text(

'¡Entorno configurado y listo para programar!',

style: TextStyle(fontSize: 20),

),

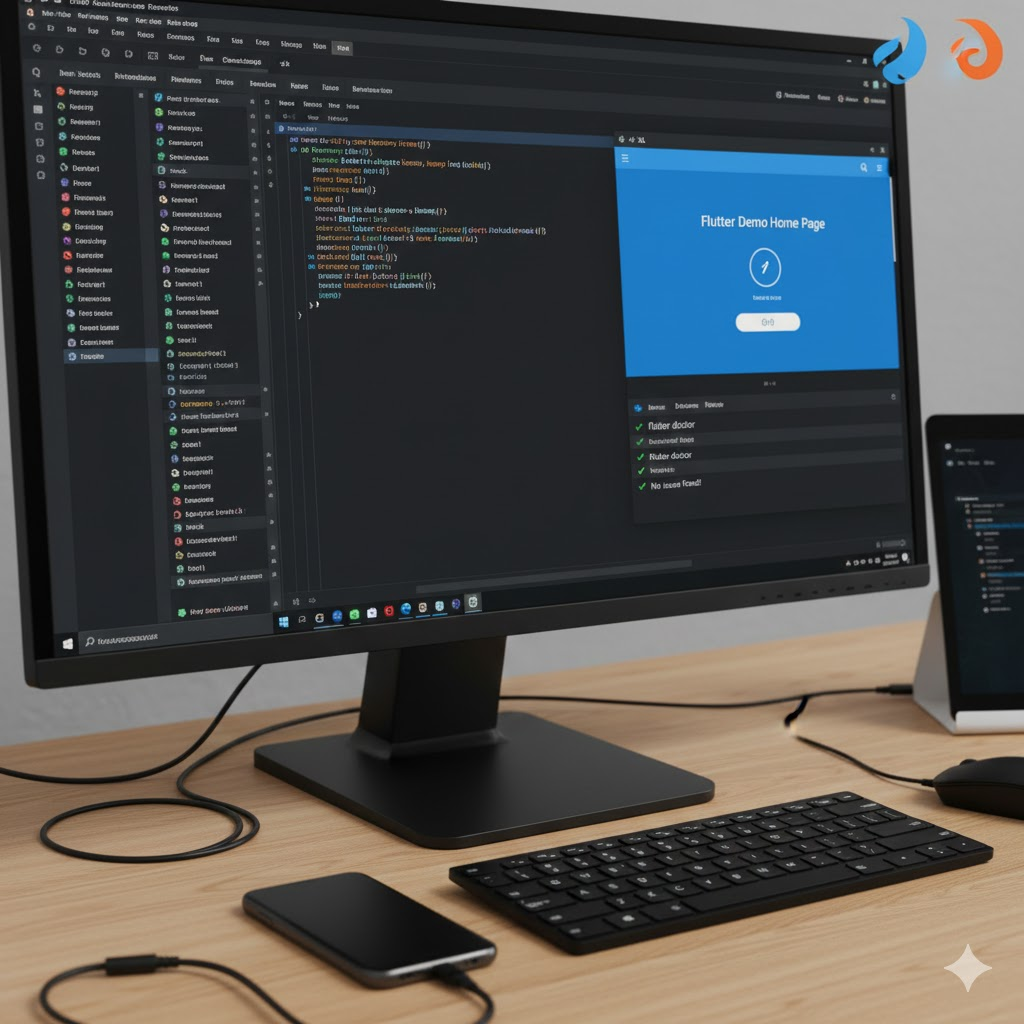
),

),

);

}

}



# Semana 2: Fundamentos de Dart

**1. Introducción a la Semana**

En esta semana el estudiante se familiariza con todo el entorno necesario para comenzar a desarrollar aplicaciones en Flutter. El objetivo principal es asegurarse de que cada alumno tenga un ambiente de trabajo correctamente configurado, comprenda cómo estructurar un proyecto y pueda ejecutar su primera aplicación básica.

**2. Instalación del Entorno de Desarrollo**

Durante esta etapa se revisa cada uno de los componentes requeridos para trabajar con Flutter. Se explica qué es el SDK de Flutter, para qué sirve Dart y cómo se integran ambas herramientas dentro del proceso de desarrollo.

**Puntos clave trabajados:**

* Instalación del **SDK de Flutter** en Windows, macOS o Linux.
* Configuración de rutas del sistema para que Flutter pueda ejecutarse desde la terminal.
* Instalación de un **editor de código**, principalmente Visual Studio Code o Android Studio.
* Revisión de extensiones necesarias, como Flutter y Dart.
* Configuración de emuladores de Android o dispositivos físicos.

Se practican comandos básicos como flutter doctor, que permite verificar que todo esté correctamente instalado y detectar posibles fallos en la configuración.

**3. Creación del Primer Proyecto en Flutter**

Una vez configurado el entorno, se enseña a generar un nuevo proyecto desde la terminal o desde el IDE.

**Se aprende a:**

* Crear un proyecto base.
* Explorar los archivos generados automáticamente.
* Comprender la estructura del árbol de carpetas.

Se explica el rol de archivos fundamentales como:

* pubspec.yaml: gestión de dependencias.
* Carpeta lib: espacio donde se escribe la lógica de la aplicación.
* Carpeta android y ios: configuración específica para cada sistema.

**4. Ejecución de la Aplicación por Primera Vez**

El estudiante aprende a iniciar un emulador o conectar un dispositivo físico y ejecutar la aplicación generada.

**Actividades realizadas:**

* Ejecutar la aplicación en modo debug.
* Observar en tiempo real cómo Flutter compila los cambios mediante Hot Reload y Hot Restart.
* Detectar errores comunes y su solución básica.

La finalidad es que cada alumno obtenga una aplicación funcional en pantalla, aunque sea la versión predeterminada que proporciona Flutter.

**5. Comprensión de la Estructura del Código Inicial**

En esta parte se analiza el archivo principal que aparece en el proyecto recién creado.

Se explican conceptos como:

* main(): punto de entrada de una aplicación Flutter.
* runApp(): función encargada de iniciar la interfaz.
* Widgets básicos utilizados en el proyecto inicial.
* Cómo está organizada la interfaz mediante widgets padres e hijos.

El propósito es que el estudiante entienda cómo funciona internamente una aplicación Flutter antes de escribir código propio.

| **Tema Principal** | **Conceptos Clave** | **Sintaxis Clave de Dart** |
| --- | --- | --- |
| **Tipos de Datos y Variables** | **Tipado Fuerte:** Dart infiere el tipo, pero es seguro. Tipos básicos: int, double, String, bool, List, Map. **final vs. const:** const es inmutable en tiempo de compilación; final es inmutable después de la inicialización. | int edad = 30;, String nombre = 'Ana';, final fecha = DateTime.now();, const PI = 3.14; |
| **Estructuras de Control** | Uso estándar de if/else, switch, y bucles (for, while). Concepto de **Null Safety** (? y !) para manejar valores nulos de forma segura, evitando errores comunes de runtime. | int? contador;, if (condicion) {}, for (var item in lista) {} |
| **Funciones y POO** | **Funciones:** Obligatorias (posicionales) y opcionales (nombradas con {}). **Clases:** Herencia, interfaces implícitas, **constructores** (nombradores, const). | void funcion(String req, {int? opt}) {}, class Persona extends Objeto {}, this.propiedad. |

**Fundamentos de Dart**

**Foco Adicional:** **Null Safety y Programación Asíncrona.**

* **Null Safety:** Se profundiza en el uso de operadores:
  + ?: Indica que una variable puede ser nula (int? contador).
  + !: Operador "Bang", afirma que un valor no es nulo (usar con precaución, solo si estás seguro de que el valor existe).
  + ??: Operador de coalescencia nula, proporciona un valor por defecto si el valor original es nulo (edad ?? 18).
* **Asincronía (Futures):** Todo el consumo de APIs se basa en Future<T>. Los desarrolladores deben entender que las funciones async devuelven automáticamente un Future y que await pausa la ejecución *dentro* de la función async hasta que el Future se completa.

# EJEMPLO:

void main() {

// 1. Tipos básicos y null safety

int? edad; // Puede ser nulo

String nombre = 'Wendoline'; // No puede ser nulo

// Asignación con chequeo de nulo

if (edad == null) {

edad = 25;

}

// 2. Uso de una función

saludar(nombre: nombre, edad: edad);

// 3. Uso de coalescencia nula (valor por defecto)

print('Edad final: ${edad ?? 0}');

}

// Función con parámetros nombrados opcionales

void saludar({required String nombre, int? edad, String puesto = 'Desarrolladora'}) {

print('Hola, $nombre! Tu puesto es $puesto.');

// Uso del operador '!' solo si estamos seguros de que 'edad' no es nulo.

if (edad != null && edad > 18) {

print('Tienes ${edad!} años y eres mayor de edad.');

}

}



# Semana 3: Widgets básicos de Flutter

**Introducción al tema**

Durante la semana 3, el estudiante se centra en conocer y manejar los widgets básicos de Flutter, que son los bloques fundamentales para construir la interfaz de cualquier aplicación. Se comprende que en Flutter todo elemento visual es un widget, desde un simple texto hasta estructuras más complejas, y que cada widget puede personalizarse mediante propiedades para modificar su apariencia y comportamiento. Esta semana permite al estudiante familiarizarse con la creación de interfaces simples y la organización de widgets en jerarquías.

**Widgets fundamentales**

Se estudian widgets esenciales como **MaterialApp**, **Scaffold**, **AppBar**, **Text**, **Container**, **Row** y **Column**. Se explica la función de cada uno: MaterialApp como contenedor principal de la aplicación, Scaffold para organizar la estructura básica de la pantalla, AppBar para encabezados, Text para mostrar información, y Row y Column para distribuir elementos horizontal o verticalmente. Se enseña cómo combinar estos widgets para crear interfaces completas y ordenadas.

**Manejo de imágenes e íconos**

Durante la semana se trabaja con widgets de imagen e íconos, incluyendo **Image.asset**, **Image.network** e **Icon**. Se aprende a incluir imágenes locales en la carpeta assets y configurarlas en el archivo **pubspec.yaml** para que la aplicación las reconozca. También se utilizan **IconButton** y otros widgets interactivos, permitiendo que la interfaz tenga elementos funcionales y visualmente atractivos.

**Jerarquía y estructura de widgets**

Se explica cómo los widgets se organizan en una jerarquía padre-hijo. El estudiante entiende que la forma de combinar widgets determina el diseño de la interfaz y cómo se comportan los elementos en la pantalla. Se enfatiza la importancia de planear la disposición de los widgets y utilizar contenedores como **Container** para agruparlos y aplicar estilos, colores y bordes.

**Creación de una pantalla básica**

Durante la práctica, el estudiante crea una pantalla sencilla que incluye un encabezado, un área central con texto y un botón. Se aplica todo lo aprendido sobre widgets y jerarquía, y se observa cómo los cambios en el código afectan la visualización gracias a la función **Hot Reload**. Esta actividad refuerza la comprensión de los conceptos teóricos y permite adquirir habilidades prácticas en la construcción de interfaces en Flutter.

**Ejecución y validación de la interfaz**

Finalmente, se ejecuta la aplicación en un emulador o dispositivo físico. El estudiante verifica que todos los widgets funcionen correctamente, que las imágenes se muestren de manera adecuada y que los botones respondan a la interacción. Con esto se completa la semana, dejando al estudiante preparado para trabajar con formularios, entradas de usuario y navegación en semanas posteriores.

| **Tema Principal** | **Conceptos Clave** | **Widgets Esenciales** |
| --- | --- | --- |
| **Filosofía de Widgets** | **Todo es un Widget.** Los widgets son inmutables. La interfaz se construye como un **Árbol de Widgets**. | **MaterialApp:** Contenedor raíz (temas, rutas). **Scaffold:** Estructura de pantalla (AppBar, Body). |
| **Estructura de Layout** | **Widgets de Contenedor:** Utilizados para agrupar, aplicar estilos (color, padding, margin) y restricciones de tamaño. **Widgets de Posición:** Usados para el diseño principal. | **Container**, **Text**, **Icon**, **Row** (horizontal), **Column** (vertical), **Center**. |
| **Row y Column** | **Ejes:** mainAxisAlignment (ej. SpaceBetween) y crossAxisAlignment (ej. Center). Es crucial entender cómo el espacio es distribuido. | Row(mainAxisAlignment: ...) |

**Widgets Básicos de Flutter**

**Foco Adicional:** **Limitaciones de Row/Column y el uso de Expanded.**

* **Anidamiento y Profundidad:** Se recomienda usar la extensión **Flutter Outline** en el IDE para visualizar el árbol de widgets, ya que el anidamiento (poner un Column dentro de otro Container, etc.) es inevitable y crucial para el diseño.
* **Manejo de Espacio:** Si se coloca un widget con restricciones de tamaño indefinidas (ej. Image.network) directamente dentro de un Row o Column, se generará un **Overflow Error**. La solución es envolver el widget en un Expanded o Flexible para que ocupe el espacio restante sin desbordarse.
* **Separación de Código:** Es una buena práctica extraer widgets grandes (Scaffold o secciones complejas) en clases separadas (.dart files) para mejorar la legibilidad.

# EJEMPLO:

class LayoutBasico extends StatelessWidget {

const LayoutBasico({super.key});

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(title: const Text('Semana 3: Row y Column')),

body: Center(

child: Column(

mainAxisAlignment: MainAxisAlignment.spaceAround, // Espacio alrededor

crossAxisAlignment: CrossAxisAlignment.center, // Centra horizontalmente

children: <Widget>[

const Text('Encabezado de la Columna'),

Row(

mainAxisAlignment: MainAxisAlignment.center, // Centra horizontalmente el Row

children: const <Widget>[

Icon(Icons.star, color: Colors.amber),

SizedBox(width: 10), // Espacio entre elementos

Text('Elementos en fila'),

],

),

Container(

color: Colors.blueGrey,

padding: const EdgeInsets.all(15.0),

child: const Text('Contenedor con estilo', style: TextStyle(color: Colors.white)),

),

],

),

),

);

}

}

# Semana 4: Manejo de imágenes e íconos

**Introducción al tema**

Durante la semana 4, el estudiante se enfoca en aprender a manejar imágenes e íconos dentro de las aplicaciones Flutter. Se comprende que estos elementos son esenciales para mejorar la experiencia visual del usuario y darle identidad a la aplicación. El objetivo de la semana es que el estudiante pueda agregar imágenes locales y desde internet, así como utilizar íconos predefinidos o personalizados dentro de la interfaz.

**Uso de imágenes locales y de red**

Se enseña al estudiante a utilizar widgets como **Image.asset** para mostrar imágenes que se encuentran dentro del proyecto, y **Image.network** para mostrar imágenes alojadas en internet. Se explica cómo organizar los archivos de imagen en la carpeta **assets** y cómo configurarlos correctamente en el archivo **pubspec.yaml**, para que Flutter los reconozca durante la ejecución de la aplicación. También se muestra la importancia de optimizar las imágenes para no afectar el rendimiento de la app.

**Uso de íconos y botones con íconos**

El estudiante aprende a utilizar widgets como **Icon** y **IconButton** para agregar íconos visuales e interactivos a la interfaz. Se explica cómo modificar sus propiedades, como tamaño, color y alineación, y cómo combinar íconos con botones para crear elementos que respondan a la interacción del usuario. Esto permite que la interfaz sea más intuitiva y visualmente atractiva.

**Jerarquía y organización visual**

Se enfatiza la importancia de la jerarquía de widgets al trabajar con imágenes e íconos, asegurando que se ubiquen correctamente dentro de contenedores como **Container**, **Row** y **Column**. El estudiante comprende cómo los widgets padres influyen en la posición y el tamaño de los widgets hijos, y cómo combinar estos elementos para crear un diseño armónico y funcional.

**Creación de una interfaz con imágenes e íconos**

Durante la práctica, el estudiante desarrolla una pantalla que integra imágenes locales y de red, así como íconos interactivos. Se aplica todo lo aprendido sobre jerarquía, propiedades de widgets y organización visual. Esta actividad permite reforzar la comprensión teórica mediante la implementación práctica de los conceptos, logrando una interfaz funcional y estéticamente agradable.

**Ejecución y validación de la interfaz**

Finalmente, se ejecuta la aplicación en un emulador o dispositivo físico para verificar que todas las imágenes e íconos se muestren correctamente y que los botones respondan a la interacción del usuario. Esta actividad permite al estudiante asegurar que la interfaz funciona de manera correcta y preparar las bases para temas posteriores como navegación entre pantallas y manejo de formularios.

| **Tema Principal** | **Conceptos Clave** | **Widgets y Configuración** |
| --- | --- | --- |
| **Imágenes Locales (Assets)** | Las imágenes deben estar en una carpeta designada (assets/). El directorio debe ser declarado en el archivo **pubspec.yaml** para que Flutter pueda acceder a ellas. | **Image.asset('ruta/imagen.png')** |
| **Imágenes desde Red** | Se utilizan para contenido dinámico (ej. fotos de perfil). El acceso a la red requiere permiso en la configuración nativa (por defecto, Flutter lo maneja). | **Image.network('url')** |
| **Íconos y Botones** | Flutter usa el set de **Material Icons**. Los íconos se pueden combinar con botones para crear elementos interactivos claros. | **Icon(Icons.name)**, **IconButton**, **Image.network** |

**Manejo de Imágenes e Íconos**

**Foco Adicional:** **Resolución de Imágenes y pubspec.yaml.**

* **DPI/Densidad:** Se introduce el concepto de **Densidad de Píxeles (DPI)**. Para optimizar imágenes en Flutter, se recomienda usar diferentes carpetas para distintas densidades: assets/1.0x/, assets/2.0x/, assets/3.0x/. Flutter selecciona automáticamente la imagen de mejor resolución para el dispositivo.
* **Image Fit:** Uso de la propiedad fit en el widget Image (BoxFit.cover, BoxFit.contain) para controlar cómo la imagen llena el espacio asignado por su widget padre.
* **pubspec.yaml:** La declaración de la carpeta assets debe ser precisa, incluyendo el slash final: assets: - assets/images/

# EJEMPLO:

class ImagenesDemo extends StatelessWidget {

const ImagenesDemo({super.key});

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(title: const Text('Semana 4: Imágenes e Íconos')),

body: ListView( // Usamos ListView para poder hacer scroll

padding: const EdgeInsets.all(20),

children: <Widget>[

const Text('Icono de Flutter:', style: TextStyle(fontSize: 18)),

const Icon(

Icons.mobile\_friendly,

size: 50,

color: Colors.teal,

),

const SizedBox(height: 20),

const Text('Imagen desde la Red:', style: TextStyle(fontSize: 18)),

// Una URL de imagen de prueba

Image.network(

'https://picsum.photos/400/200',

height: 200,

fit: BoxFit.cover,

),

const SizedBox(height: 20),

const Text('Botón con Ícono:', style: TextStyle(fontSize: 18)),

IconButton(

icon: const Icon(Icons.send),

color: Colors.indigo,

iconSize: 30,

onPressed: () {

// Lógica al presionar

print('Botón presionado');

},

),

],

),

);

}

}



# Semana 5: Navegación entre pantallas

**Introducción al tema**

Durante la semana 5, el estudiante aprende a implementar la navegación entre diferentes pantallas dentro de una aplicación Flutter. Se comprende que la navegación es fundamental para crear aplicaciones más complejas y funcionales, donde el usuario puede moverse entre secciones o funcionalidades de manera fluida.

**Rutas y Navigator**

Se estudia el uso del widget **Navigator**, que permite gestionar la pila de pantallas de la aplicación mediante los métodos **push** y **pop**. Se explica cómo crear rutas nombradas para organizar la navegación y cómo pasar información entre pantallas mediante parámetros. Esto permite que la aplicación tenga un flujo de interacción más estructurado y fácil de mantener.

**Jerarquía y organización de widgets en navegación**

Se enfatiza la importancia de mantener una jerarquía clara de widgets en cada pantalla, asegurando que los elementos principales estén organizados correctamente y que la interfaz sea consistente al navegar de una pantalla a otra. Se analiza cómo cada pantalla puede contener widgets similares o personalizados, dependiendo de su funcionalidad.

**Creación de pantallas y flujo de navegación**

Durante la práctica, el estudiante crea al menos tres pantallas conectadas entre sí. Se implementan botones que permiten moverse hacia adelante y hacia atrás, y se prueba el paso de información entre las pantallas. Esta actividad ayuda a entender cómo se construye una estructura de navegación básica, que es esencial para aplicaciones funcionales y completas.

**Ejecución y validación de la navegación**

Finalmente, se ejecuta la aplicación en un emulador o dispositivo físico para comprobar que todas las pantallas se muestren correctamente, que los botones funcionen como se espera y que la información se transfiera adecuadamente. Esta actividad permite al estudiante verificar que la navegación de la aplicación es fluida y prepara el terreno para el manejo del estado y la interactividad avanzada.

| **Tema Principal** | **Conceptos Clave** | **Flujo de Vida y Métodos** |
| --- | --- | --- |
| **Stateless vs. Stateful** | **StatelessWidget:** No tiene estado interno, es estático. **StatefulWidget:** Contiene un objeto State que puede ser modificado (dinámico). | Separación de **Widget** (la configuración) y **State** (los datos cambiantes). |
| **Manejo del Estado** | El estado son los datos que cambian con el tiempo o la interacción. El método **setState()** es la única forma de notificar a Flutter que un widget ha cambiado y debe ser redibujado (rebuild). | **setState(() { ... })** |
| **Ciclo de Vida** | Define la secuencia de métodos llamados durante la existencia de un widget. | **initState():** Se llama una vez al crear el State. **build():** Se llama siempre que se necesita redibujar. **dispose():** Se llama justo antes de que el widget sea eliminado de la pila. |

**Navegación entre Pantallas**

**Foco Adicional:** **Navegación con MaterialPageRoute y Nombres de Rutas.**

* **Rutas Nombradas (Ventajas):** Aunque push() con MaterialPageRoute funciona para casos simples, las rutas nombradas son esenciales en aplicaciones grandes para:
  + **Limpieza de Código:** Evitar referencias circulares entre archivos.
  + **Routing Dinámico:** Implementar Deep Links o rutas condicionales de forma más sencilla.
* **Rutas de Reemplazo:** Uso de Navigator.pushReplacementNamed() para reemplazar la pantalla actual y evitar que el usuario vuelva atrás (común después del inicio de sesión).
* **onGenerateRoute:** Un mecanismo avanzado para manejar rutas dinámicas que contienen parámetros en la URL (ej. /detalles/123).

# EJEMPLO:

// main.dart (Definición de rutas)

void main() {

runApp(MaterialApp(

initialRoute: '/',

routes: {

'/': (context) => const PantallaA(),

'/segunda': (context) => const PantallaB(),

},

));

}

// Pantalla A: Envía al usuario a la segunda pantalla

class PantallaA extends StatelessWidget {

const PantallaA({super.key});

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(title: const Text('Semana 5: Pantalla A')),

body: Center(

child: ElevatedButton(

onPressed: () {

// Navega a la ruta nombrada '/segunda'

Navigator.pushNamed(

context,

'/segunda',

arguments: 'Datos de la Pantalla A', // Pasa argumentos

);

},

child: const Text('Ir a Pantalla B'),

),

),

);

}

}

// Pantalla B: Recibe datos y permite volver

class PantallaB extends StatelessWidget {

const PantallaB({super.key});

@override

Widget build(BuildContext context) {

// Recupera los argumentos pasados

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

return Scaffold(

appBar: AppBar(title: const Text('Pantalla B')),

body: Center(

child: Column(

mainAxisAlignment: MainAxisAlignment.center,

children: [

Text('Recibido: $args', style: const TextStyle(fontSize: 18)),

ElevatedButton(

onPressed: () {

// Vuelve a la pantalla anterior

Navigator.pop(context);

},

child: const Text('Volver a Pantalla A'),

),

],

),

),

);

}

}

# Semana 6: Stateful Widgets y manejo del estado

**Introducción al tema**

Durante la semana 6, el estudiante aprende sobre los **StatefulWidgets**, que permiten crear interfaces dinámicas capaces de actualizarse en tiempo real. Se entiende que muchos elementos de una aplicación requieren interacción con el usuario y cambios constantes en la interfaz, y para esto es necesario manejar correctamente el estado de los widgets.

**Stateful vs Stateless Widgets**Se estudia la diferencia entre **StatelessWidget**, que es inmutable y no cambia durante la ejecución, y **StatefulWidget**, que puede cambiar y actualizar su contenido según la interacción del usuario o los datos recibidos. Se analiza cuándo usar cada tipo de widget y cómo estructurarlos dentro de la aplicación.

**Uso de setState y ciclo de vida**

El estudiante aprende a utilizar el método **setState()** para actualizar la interfaz de manera controlada. También se explican conceptos básicos del ciclo de vida de un StatefulWidget, incluyendo **initState**, **build** y **dispose**, lo que permite comprender cómo se inicializan, renderizan y eliminan los widgets dinámicos.

**Creación de interfaces interactivas**

Durante la práctica, el estudiante desarrolla un contador interactivo que puede aumentar, disminuir y reiniciarse mediante botones. Esta actividad permite aplicar los conceptos de StatefulWidget y setState, y observar cómo la interfaz se actualiza automáticamente al modificar los datos. Además, se prueba la combinación de StatefulWidgets con widgets básicos como Text, Row, Column y Container.

**Ejecución y validación de la interactividad**

Finalmente, la aplicación se ejecuta en un emulador o dispositivo físico para verificar que los cambios en los valores se reflejen correctamente en la pantalla. El estudiante confirma que los widgets responden a la interacción del usuario, que la actualización de la interfaz funciona correctamente y que la aplicación mantiene un comportamiento estable durante las pruebas. Esta semana establece las bases para trabajar con formularios, listas dinámicas y manejo avanzado de datos en Flutter.

| **Tema Principal** | **Conceptos Clave** | **Flujo de Vida y Métodos** |
| --- | --- | --- |
| **Stateless vs. Stateful** | **StatelessWidget:** No tiene estado interno, es estático. **StatefulWidget:** Contiene un objeto State que puede ser modificado (dinámico). | Separación de **Widget** (la configuración) y **State** (los datos cambiantes). |
| **Manejo del Estado** | El estado son los datos que cambian con el tiempo o la interacción. El método **setState()** es la única forma de notificar a Flutter que un widget ha cambiado y debe ser redibujado (rebuild). | **setState(() { ... })** |
| **Ciclo de Vida** | Define la secuencia de métodos llamados durante la existencia de un widget. | **initState():** Se llama una vez al crear el State. **build():** Se llama siempre que se necesita redibujar. **dispose():** Se llama justo antes de que el widget sea eliminado de la pila. |

**Stateful Widgets y Manejo del Estado**

**Foco Adicional:** **La Inmutabilidad y el Costo del setState.**

* **Inmutabilidad del Widget:** Se refuerza que el StatefulWidget en sí es inmutable, pero el objeto State (donde se almacenan las variables) es mutable.
* **Alcance de setState:** setState() reconstruye *todo* el método build() del widget que lo contiene y de sus descendientes. Si un widget es muy grande o contiene muchas ramas, esto puede ser ineficiente. Por ello, es clave mantener los StatefulWidgets lo más pequeños posible (el concepto de **State Locality**).
* **Ciclo de Vida Extendido:**
  + didChangeDependencies(): Se llama cuando el objeto State cambia su dependencia de otro objeto (ej. el Theme).
  + didUpdateWidget(): Se llama cuando el widget padre reconstruye el widget actual con nuevos parámetros.

# EJEMPLO:

class ContadorDemo extends StatefulWidget {

const ContadorDemo({super.key});

@override

State<ContadorDemo> createState() => \_ContadorDemoState();

}

class \_ContadorDemoState extends State<ContadorDemo> {

// Estado: Variable que queremos que cambie en la UI

int \_contador = 0;

void \_incrementar() {

// Llama a setState() para notificar a Flutter que el estado cambió

setState(() {

\_contador++;

});

}

void \_decrementar() {

setState(() {

\_contador--;

});

}

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(title: const Text('Semana 6: Contador Stateful')),

body: Center(

child: Column(

mainAxisAlignment: MainAxisAlignment.center,

children: <Widget>[

const Text('Valor actual:', style: TextStyle(fontSize: 20)),

Text(

'$\_contador',

style: const TextStyle(fontSize: 48, fontWeight: FontWeight.bold, color: Colors.blue),

),

const SizedBox(height: 20),

Row(

mainAxisAlignment: MainAxisAlignment.center,

children: [

FloatingActionButton(

onPressed: \_incrementar,

tooltip: 'Incrementar',

child: const Icon(Icons.add),

),

const SizedBox(width: 20),

FloatingActionButton(

onPressed: \_decrementar,

tooltip: 'Decrementar',

child: const Icon(Icons.remove),

),

],

),

],

),

),

);

}

}

# Semana 7: Formularios e inputs

**Introducción al tema**

Durante la semana 7, el estudiante aprende a trabajar con formularios y entradas de datos dentro de una aplicación Flutter. Se comprende que capturar información del usuario es fundamental para desarrollar aplicaciones funcionales, como registros, inicios de sesión o encuestas. Se estudian las herramientas que Flutter ofrece para validar y procesar esta información de manera efectiva.

**Widgets para formularios y entradas**

Se estudian widgets como **TextField** y **TextFormField**, que permiten capturar información del usuario. Se aprende a configurar sus propiedades para personalizar el estilo, tipo de teclado, texto de ayuda y control de errores. Además, se introducen botones para enviar o limpiar los datos ingresados por el usuario.

**Validación de datos**

El estudiante aprende a implementar validaciones básicas, como verificar que un campo no esté vacío, que un correo tenga el formato correcto o que un número cumpla ciertas condiciones. Se explican las funciones necesarias para realizar estas comprobaciones y cómo mostrar mensajes de error al usuario cuando la información ingresada no es válida.

**Creación de formularios funcionales**

Durante la práctica, se desarrolla un formulario completo de registro que incluye campos de texto, correo electrónico y contraseña. Se agregan botones para enviar la información y mensajes de validación que ayudan al usuario a completar correctamente los campos. Esta actividad permite al estudiante aplicar los conceptos teóricos en un ejemplo práctico y funcional.

**Ejecución y validación del formulario**

Finalmente, la aplicación se ejecuta en un emulador o dispositivo físico para comprobar que todos los campos funcionan correctamente, que los mensajes de validación se muestran cuando es necesario y que la información ingresada puede ser procesada o almacenada según los requerimientos de la aplicación. Esta semana establece la base para trabajar con listas dinámicas y consumo de APIs

| **Tema Principal** | **Conceptos Clave** | **Widgets y Controles** |
| --- | --- | --- |
| **Entradas de Texto** | Se utilizan para capturar datos del usuario (texto, contraseñas, números). Se controlan mediante **TextEditingController** para leer o modificar el valor programáticamente. | **TextField**, TextFormField, **TextEditingController**. |
| **Validación de Datos** | La validación se realiza para asegurar que los datos cumplen un formato requerido. El widget **Form** es crucial para agrupar y validar múltiples campos a la vez. | **Form**, GlobalKey para acceder al estado del formulario, **validator: (value) => ...** |
| **Experiencia de Usuario** | Propiedades para mejorar la interacción: tipo de teclado (keyboardType), texto de ayuda (decoration: InputDecoration), seguridad (obscureText). | keyboardType, decoration, obscureText. |

**Formularios e Inputs**

**Foco Adicional:** **Controladores y Persistencia de Datos.**

* **TextEditingController:** El uso de controladores es fundamental no solo para **leer** el valor del campo, sino también para **establecer** un valor inicial (ej. en un formulario de edición) o **limpiar** el campo después de un envío exitoso.
* **Form vs. TextFormField:** Mientras que TextField es simple, TextFormField está diseñado para funcionar con el widget Form, ya que Form proporciona el método validate() que ejecuta el validator de *todos* los TextFormField hijos.
* **Manejo de Foco:** Se introduce el uso de FocusNode para controlar qué campo tiene el cursor activo, lo que es útil para pasar al siguiente campo automáticamente al presionar "Enter

# EJEMPLO:

class FormularioDemo extends StatefulWidget {

const FormularioDemo({super.key});

@override

State<FormularioDemo> createState() => \_FormularioDemoState();

}

class \_FormularioDemoState extends State<FormularioDemo> {

// 1. Clave para identificar y validar el Form

final \_formKey = GlobalKey<FormState>();

String \_nombre = '';

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(title: const Text('Semana 7: Validación de Formulario')),

body: Padding(

padding: const EdgeInsets.all(20.0),

child: Form(

key: \_formKey, // Asigna la clave

child: Column(

children: <Widget>[

TextFormField(

decoration: const InputDecoration(labelText: 'Nombre'),

// 2. Función de validación

validator: (value) {

if (value == null || value.isEmpty) {

return 'El nombre no puede estar vacío';

}

\_nombre = value;

return null; // Retorna null si la validación es exitosa

},

),

const SizedBox(height: 20),

ElevatedButton(

onPressed: () {

// 3. Verifica el estado actual de la validación

if (\_formKey.currentState!.validate()) {

ScaffoldMessenger.of(context).showSnackBar(

SnackBar(content: Text('Formulario Enviado. Nombre: $\_nombre')),

);

}

},

child: const Text('Enviar Datos'),

),

],

),

),

),

);

}

}

# Semana 8: Listas y manejo de scroll

**Introducción al tema**

Durante la semana 8, el estudiante se enfoca en el manejo de listas y desplazamiento dentro de la interfaz de Flutter. Se comprende que la presentación de información en listas es una de las estructuras más comunes en las aplicaciones móviles, y que es necesario dominar cómo crear, mostrar y actualizar estos elementos dinámicamente.

**Widgets para listas**

Se estudian widgets como **ListView**, **ListTile** y **ListView.builder**, que permiten mostrar elementos en forma de lista y manejar el scroll de manera automática. Se aprende a organizar los elementos de la lista y a personalizar su apariencia para mejorar la experiencia del usuario.

**Listas dinámicas y actualización de datos**

El estudiante aprende a crear listas dinámicas que pueden aumentar o disminuir según la interacción del usuario. Se implementa la capacidad de agregar, eliminar o modificar elementos de manera interactiva, utilizando métodos y StatefulWidgets para actualizar la interfaz en tiempo real.

**Creación de listas funcionales**

Durante la práctica, se desarrolla una pantalla que muestra una lista de tareas o productos. Se agregan botones para añadir nuevos elementos, eliminar existentes y actualizar los datos de cada ítem. Esta actividad permite al estudiante aplicar los conceptos de widgets dinámicos y manejo de estado en un ejemplo práctico.

**Ejecución y validación de las listas**

Finalmente, se ejecuta la aplicación en un emulador o dispositivo físico para verificar que la lista se despliegue correctamente, que el scroll funcione de manera fluida y que los elementos puedan agregarse, eliminarse o actualizarse sin errores. Con esto, el estudiante queda preparado para trabajar con consumo de APIs y mostrar datos externos en la aplicación.

| **Tema Principal** | **Conceptos Clave** | **Widgets para Listas** |
| --- | --- | --- |
| **Listas Estáticas vs. Dinámicas** | Listas estáticas (Column o ListView simple) son ineficientes para muchos elementos. Se requieren constructores para listas dinámicas basadas en datos. | **ListView**, **ListTile** (elemento simple de lista). |
| **ListView.builder** | Permite construir elementos de la lista *solo cuando son visibles en la pantalla*, lo que optimiza el rendimiento y el consumo de memoria, esencial para listas grandes. | **ListView.builder(itemCount: ..., itemBuilder: ...)** |
| **Gestión de Datos** | Las listas se actualizan modificando el arreglo (List<T>) subyacente y llamando a setState() para que el ListView.builder se redibuje con el nuevo itemCount. | List.add(), List.remove(), **setState()** |

**Listas y Manejo de Scroll**

**Foco Adicional:** **Tipos de ListView y la Interfaz Sliver.**

* **ListView Tipos:**
  + ListView(): Lista simple de todos los hijos (estático).
  + ListView.builder(): El método de construcción a demanda (dinámico y optimizado).
  + ListView.separated(): Similar al builder, pero añade un separador entre cada ítem.
* **Scroll Direccional:** Uso de la propiedad scrollDirection: Axis.horizontal en ListView para crear carruseles o listas horizontales.
* **Slivers (Concepto Avanzado):** Se menciona que ListView es en realidad un SliverList. Los **Slivers** son widgets que controlan la porción del área de scroll que ocupan. Esto es clave para efectos de scroll avanzados, como el **SliverAppBar** que se colapsa.

# EJEMPLO:

class ListaDinamicaDemo extends StatelessWidget {

const ListaDinamicaDemo({super.key});

@override

Widget build(BuildContext context) {

// Genera una lista de 100 elementos de prueba

final List<String> items = List.generate(100, (i) => 'Elemento N° ${i + 1}');

return Scaffold(

appBar: AppBar(title: const Text('Semana 8: ListView.builder')),

// 1. Usa ListView.builder para renderizar solo los visibles

body: ListView.builder(

itemCount: items.length, // Total de elementos

itemBuilder: (context, index) {

// 2. El widget que se construye para cada elemento

return ListTile(

leading: const Icon(Icons.list),

title: Text(items[index]),

subtitle: Text('Índice: $index'),

onTap: () {

print('Clic en ${items[index]}');

},

);

},

),

);

}

}

# Semana 9: Consumo de APIs (Parte 1: HTTP y JSON)

**Introducción al tema**

Durante la semana 9, el estudiante se introduce al consumo de **APIs** dentro de una aplicación Flutter, comprendiendo que muchas aplicaciones modernas dependen de servicios externos para obtener información dinámica. Se explica que una API (Application Programming Interface) permite la comunicación entre la aplicación y un servidor para enviar y recibir datos, generalmente en formato JSON, que luego pueden ser procesados y mostrados en la interfaz de la app.

El objetivo de la semana es que el estudiante entienda cómo establecer conexiones a internet desde Flutter, cómo enviar solicitudes y cómo recibir y procesar la información de manera efectiva, sin afectar el rendimiento de la aplicación ni la experiencia del usuario.

**Uso del paquete HTTP**

Se enseña al estudiante a utilizar el paquete **http**, que permite realizar solicitudes **GET** y **POST** a servidores externos. Se analiza cómo importar correctamente el paquete en el proyecto, cómo construir la URL de la solicitud y cómo manejar la respuesta del servidor. También se explica la importancia de manejar posibles errores, como la falta de conexión, tiempo de espera excedido o respuestas inesperadas.

**Manejo de JSON**

Una vez obtenida la respuesta de la API, el estudiante aprende a decodificar los datos con **json.decode**, transformando el contenido en objetos o listas de Dart que puedan ser utilizados dentro de la aplicación. Se estudia cómo acceder a cada campo del JSON, recorrer listas de objetos y extraer la información necesaria para mostrarla posteriormente en la interfaz.

**Creación de la práctica con API**

Durante la práctica, el estudiante realiza una solicitud a una API pública, como una de usuarios o publicaciones de prueba. Inicialmente, los datos obtenidos se muestran en la consola para verificar que la conexión y la decodificación sean correctas. Posteriormente, se planifica cómo estos datos se integrarán en la interfaz de la aplicación en semanas posteriores. Esta práctica permite al estudiante comprender todo el flujo de información, desde la solicitud hasta el procesamiento de los datos.

**Ejecución y validación de la práctica**

Finalmente, se ejecuta la aplicación en un emulador o dispositivo físico para verificar que la conexión funcione correctamente y que los datos se reciban sin errores. El estudiante asegura que la información sea coherente con la API y comprende la importancia de validar la respuesta antes de mostrarla al usuario. Esta semana sienta las bases para la visualización de los datos y la construcción de interfaces dinámicas en la siguiente etapa.

| **Tema Principal** | **Conceptos Clave** | **Paquetes y Métodos** |
| --- | --- | --- |
| **El Paquete HTTP** | Permite realizar solicitudes a servidores externos (APIs REST). El acceso a la red siempre es una **operación asincrónica**. | Paquete **http**, import 'package:http/http.dart' as http;. |
| **Solicitudes Asincrónicas** | Uso de **async y await** para manejar operaciones de red sin bloquear el hilo principal de la aplicación (UI). | Future<http.Response> fetchData() async { ... } |
| **Manejo de JSON** | La respuesta de la API (JSON string) debe ser convertida a un objeto o mapa de Dart para poder acceder a los datos. El proceso de mapeo es **Decodificación**. | **dart:convert** librería, json.decode(response.body). |

**Consumo de APIs (Parte 1: HTTP y JSON)**

**Foco Adicional:** **Modelos de Datos y Códigos de Estado HTTP.**

* **Serialización/Deserialización:** Se insiste en la importancia de crear **Modelos de Datos (Clases Dart)** que reflejen la estructura JSON de la API. Esto se conoce como **Decodificación Manual de JSON (Manejo sin librería)** y es vital para la seguridad de tipos.
* **Códigos de Estado:** Es crucial validar el código de estado de la respuesta antes de decodificar el cuerpo. Un código 200 (OK) o 201 (Created) significa éxito, mientras que 4xx (Error de Cliente) o 5xx (Error de Servidor) deben manejarse como errores.
* **Permisos de Red:** En entornos de prueba (ej. AndroidManifest.xml en Android) debe estar el permiso <uses-permission android:name="android.permission.INTERNET" />, aunque Flutter a menudo lo añade por defecto.

# EJEMPLO:

import 'package:http/http.dart' as http;

import 'dart:convert'; // Necesario para json.decode

// Definición de un modelo (Clase) simple para mapear el JSON

class Post {

final int id;

final String title;

Post({required this.id, required this.title});

}

// Función asíncrona para obtener datos

Future<List<Post>> fetchPosts() async {

// 1. Realiza la solicitud HTTP

final response = await http.get(Uri.parse('https://jsonplaceholder.typicode.com/posts'));

// 2. Verifica el código de estado

if (response.statusCode == 200) {

// 3. Decodifica el JSON de la respuesta (string -> List<Map>)

final List<dynamic> jsonList = json.decode(response.body);

// 4. Mapea los mapas decodificados a una lista de objetos Post

return jsonList.map((json) => Post(

id: json['id'] as int,

title: json['title'] as String,

)).toList();

} else {

// Lanza un error si la solicitud no fue exitosa

throw Exception('Fallo al cargar los posts: ${response.statusCode}');

}

}

# Semana 10: Consumo de APIs (Parte 2: FutureBuilder y listas)

**Introducción al tema**

Durante la semana 10, el estudiante avanza en el manejo de APIs aprendiendo a mostrar la información obtenida directamente en la interfaz de la aplicación. Se comprende que es necesario manejar los datos de forma asincrónica para que la aplicación siga siendo responsiva y que Flutter proporciona herramientas como **FutureBuilder** para construir interfaces que se actualizan automáticamente cuando los datos están disponibles.

El objetivo es que el estudiante pueda crear aplicaciones que no solo obtengan información de APIs, sino que también la presenten de manera organizada y con retroalimentación visual mientras se cargan los datos.

**Uso de FutureBuilder**

Se estudia el widget **FutureBuilder**, que permite trabajar con operaciones asincrónicas, como solicitudes HTTP, sin bloquear la interfaz. Se explican sus propiedades, como future y builder, y cómo manejar los distintos estados: cargando, datos disponibles y error. Esto permite que la aplicación muestre un indicador de progreso mientras los datos se descargan y un mensaje adecuado en caso de fallo.

**Integración de datos en listas dinámicas**

El estudiante aprende a combinar **FutureBuilder** con **ListView.builder** para mostrar la información de la API en listas dinámicas. Cada elemento de la lista se genera automáticamente a partir de los datos recibidos, permitiendo que la interfaz sea flexible y escalable. Se analizan casos como la paginación y el rendimiento de listas grandes, y cómo actualizar la interfaz al agregar, modificar o eliminar datos.

**Creación de la práctica con datos de API**

Durante la práctica, el estudiante desarrolla una pantalla que muestra la información de una API pública en forma de lista interactiva. Se agregan indicadores visuales de carga, mensajes de error y manejo de casos donde no hay datos disponibles. Esta actividad integra todo lo aprendido sobre solicitudes HTTP, decodificación de JSON y presentación de datos en la interfaz, permitiendo que la aplicación sea funcional y profesional.

**Ejecución y validación de la práctica**

Finalmente, se ejecuta la aplicación en un emulador o dispositivo físico para verificar que los datos se muestren correctamente en la pantalla, que el indicador de carga funcione mientras se espera la respuesta y que los mensajes de error se presenten de forma adecuada. El estudiante confirma que la lista se actualiza automáticamente al recibir los datos y que la aplicación mantiene un comportamiento estable. Esta semana refuerza las bases para el manejo de almacenamiento local y persistencia de datos en futuras semanas.

| **Tema Principal** | **Conceptos Clave** | **Widgets y Flujo de Datos** |
| --- | --- | --- |
| **Manejo Asincrónico de la UI** | La interfaz debe reaccionar a los 3 estados de la solicitud: **Cargando**, **Datos Disponibles**, y **Error**. | El widget **FutureBuilder** |
| **FutureBuilder** | Widget diseñado para tomar un Future<T> (ej. la solicitud HTTP) y construir la UI basándose en el estado de ese Future (si ya terminó o sigue pendiente). | **FutureBuilder(future: fetchData(), builder: (context, snapshot) { ... })** |
| **Integración con Listas** | Dentro del estado snapshot.hasData, se utiliza el dato recibido para alimentar un **ListView.builder**, creando una lista dinámica a partir de la respuesta de la API. | snapshot.data, ListView.builder. |

**Consumo de APIs (Parte 2: FutureBuilder y Listas)**

**Foco Adicional:** **Manejo de Conexiones y el Patrón Provider (Mención).**

* **Flujo del FutureBuilder:** Se debe asegurar que el future se inicialice solo una vez (generalmente en initState() o como una propiedad final), para evitar que el FutureBuilder ejecute la solicitud HTTP en cada rebuild.
* **Manejo de Errores Visuales:** En el estado snapshot.hasError, se debe mostrar un widget amigable (ej. un ícono de advertencia y un mensaje) en lugar del error técnico.
* **Gestión del Estado Asíncrono:** Se introduce el concepto de que el FutureBuilder es una solución local. Para manejar el estado de las APIs globalmente, se mencionan patrones como **Provider** o **Riverpod**, que separan la lógica de negocio de la UI.

# EJEMPLO:

// Suponiendo que la clase Post y la función fetchPosts() son accesibles

class FutureBuilderDemo extends StatelessWidget {

const FutureBuilderDemo({super.key});

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(title: const Text('Semana 10: FutureBuilder')),

// 1. El widget FutureBuilder

body: FutureBuilder<List<Post>>(

future: fetchPosts(), // La Future que queremos resolver

builder: (context, snapshot) {

// 2. Estado: Esperando (Cargando)

if (snapshot.connectionState == ConnectionState.waiting) {

return const Center(child: CircularProgressIndicator());

}

// 3. Estado: Error

if (snapshot.hasError) {

return Center(child: Text('Error: ${snapshot.error}'));

}

// 4. Estado: Datos disponibles

if (snapshot.hasData) {

final posts = snapshot.data!;

// Usa ListView.builder para mostrar la lista de Posts

return ListView.builder(

itemCount: posts.length,

itemBuilder: (context, index) {

return ListTile(

title: Text(posts[index].title),

subtitle: Text('ID: ${posts[index].id}'),

);

},

);

}

// 5. Caso por defecto (ej. No hay datos)

return const Center(child: Text('No se encontraron posts.'));

},

),

);

}

}

# Semana 11: Persistencia local con SharedPreferences

**Introducción al tema**

Durante la semana 11, el estudiante se centra en aprender a almacenar información de manera local en el dispositivo utilizando **SharedPreferences**, comprendiendo que muchas aplicaciones requieren guardar datos simples, como configuraciones, preferencias del usuario, temas visuales o información básica que debe mantenerse incluso después de cerrar la aplicación. Se explica que SharedPreferences es ideal para datos de tipo clave-valor y que su manejo es rápido y eficiente para información pequeña, sin necesidad de bases de datos complejas.

El objetivo principal de esta semana es que el estudiante pueda implementar almacenamiento local que mejore la experiencia del usuario y haga que la aplicación recuerde configuraciones y preferencias personalizadas cada vez que se abra.

**Uso de SharedPreferences**

Se estudia cómo instalar el paquete **shared\_preferences** en Flutter, importarlo correctamente en el proyecto y crear una instancia para manipular los datos. Se enseña a almacenar y recuperar distintos tipos de datos, como cadenas de texto, enteros, booleanos y listas simples, y a verificar la existencia de datos antes de mostrarlos en la interfaz. Se analiza la importancia de manejar errores al leer o escribir datos y cómo limpiar información cuando ya no es necesaria.

**Creación de la práctica**

Durante la práctica, el estudiante desarrolla una aplicación que guarda el nombre del usuario y su preferencia de tema (claro u oscuro). La aplicación permite que, al cerrar y volver a abrir, la información se recupere automáticamente y se aplique en la interfaz sin necesidad de volver a ingresar los datos. Se profundiza en cómo integrar los datos almacenados con los StatefulWidgets para actualizar la interfaz dinámicamente y reflejar los cambios de manera inmediata.

**Ejecución y validación de la práctica**

Finalmente, se ejecuta la aplicación en un emulador o dispositivo físico para comprobar que los datos se guarden y se recuperen correctamente, y que la interfaz se adapte según la información almacenada. El estudiante verifica que la aplicación funcione de manera consistente, incluso después de reiniciarla, y comprende la importancia de la persistencia local para mejorar la experiencia del usuario y la funcionalidad de la aplicación.

| **Tema Principal** | **Conceptos Clave** | **Paquetes y Uso** |
| --- | --- | --- |
| **Persistencia Local** | Almacenamiento de datos dentro del dispositivo. **SharedPreferences:** Ideal para datos **simples de clave-valor** (configuraciones, flags, preferencias). | Paquete **shared\_preferences**. |
| **Manejo de Datos** | Los datos se almacenan como pares (key: value). Solo soporta tipos primitivos (String, int, bool, double, List<String>). | **SharedPreferences.getInstance()** (asíncrono), prefs.setString('usuario', 'nombre'), prefs.getBool('darkMode'). |
| **Casos de Uso** | Guardar el estado de la sesión, las preferencias de idioma, el tema de color, o una bienvenida inicial. | Uso en initState() de un StatefulWidget para cargar las preferencias al iniciar. |

**Persistencia local con SharedPreferences**

**Foco Adicional:** **Limitaciones de SharedPreferences y la Asincronía.**

* **Asincronía de Acceso:** Se enfatiza que todos los métodos de SharedPreferences (tanto lectura como escritura) son **asíncronos** y devuelven un Future<bool> o Future<Tipo>. Esto significa que la carga inicial de preferencias debe realizarse con await en el initState().
* **Uso vs. SQLite:** Se subraya que SharedPreferences es para datos *no estructurados* y simples. **Nunca** debe usarse para almacenar grandes listas o información sensible (ej. tokens de autenticación complejos), ya que no es inherentemente seguro ni eficiente para grandes volúmenes.
* **Integración con Temas:** El ejemplo de tema claro/oscuro es el caso de uso perfecto: la preferencia de tema se guarda con prefs.setBool('isDark', true) y se lee en el inicio para aplicar el ThemeData correcto.

# EJEMPLO:

import 'package:shared\_preferences/shared\_preferences.dart';

class PreferencesDemo extends StatefulWidget {

const PreferencesDemo({super.key});

@override

State<PreferencesDemo> createState() => \_PreferencesDemoState();

}

class \_PreferencesDemoState extends State<PreferencesDemo> {

bool \_isDarkMode = false;

final String \_keyDarkMode = 'setting\_dark\_mode';

@override

void initState() {

super.initState();

// 1. Cargar la preferencia al inicio

\_loadPreferences();

}

// Función asíncrona para leer el valor

void \_loadPreferences() async {

final prefs = await SharedPreferences.getInstance();

setState(() {

// 2. Recupera el valor, usa false por defecto si es nulo

\_isDarkMode = prefs.getBool(\_keyDarkMode) ?? false;

});

}

// Función asíncrona para guardar el valor

void \_toggleDarkMode(bool value) async {

final prefs = await SharedPreferences.getInstance();

// 3. Guarda el nuevo valor

await prefs.setBool(\_keyDarkMode, value);

setState(() {

\_isDarkMode = value;

});

}

@override

Widget build(BuildContext context) {

return MaterialApp(

theme: \_isDarkMode ? ThemeData.dark() : ThemeData.light(),

home: Scaffold(

appBar: AppBar(title: const Text('Semana 11: SharedPreferences')),

body: Center(

child: Column(

mainAxisAlignment: MainAxisAlignment.center,

children: [

Text('Modo Oscuro: ${\_isDarkMode ? 'Activado' : 'Desactivado'}', style: const TextStyle(fontSize: 18)),

Switch(

value: \_isDarkMode,

onChanged: \_toggleDarkMode, // Llama a la función de guardado

),

],

),

),

),

);

}

}

# Semana 12: Bases de datos locales con SQLite

**Introducción al tema**

Durante la semana 12, el estudiante aprende a trabajar con **bases de datos locales** mediante SQLite, entendiendo que muchas aplicaciones requieren almacenar información más compleja y estructurada, como listas de tareas, notas, registros de usuarios o inventarios. Se explica que SQLite permite manejar grandes cantidades de datos de manera eficiente dentro del dispositivo, ofreciendo funcionalidades de creación, lectura, actualización y eliminación (CRUD) de registros.

El objetivo de la semana es que el estudiante sea capaz de implementar un sistema de almacenamiento local confiable, asegurando que los datos persistan y puedan ser manipulados de manera segura y organizada.

**Uso de SQLite en Flutter**

Se enseña cómo instalar el paquete **sqflite** y cómo configurarlo dentro del proyecto. El estudiante aprende a crear la base de datos, definir tablas con sus respectivos campos, y realizar operaciones CRUD. También se explica cómo manejar transacciones para mantener la integridad de los datos, cómo consultar la base de datos para obtener información específica y cómo cerrar la conexión correctamente para evitar errores o pérdida de información.

**Creación de la práctica**

Durante la práctica, el estudiante desarrolla una mini aplicación de notas o tareas, donde puede agregar nuevas entradas, visualizarlas en una lista dinámica, modificarlas y eliminarlas según sea necesario. Se combinan los conocimientos de StatefulWidgets, listas y manejo de datos para que la información se muestre en tiempo real, permitiendo al usuario interactuar con la aplicación de manera fluida y organizada.

**Ejecución y validación de la práctica**

Finalmente, se ejecuta la aplicación en un emulador o dispositivo físico para verificar que las operaciones CRUD funcionen correctamente y que los datos permanezcan almacenados de forma persistente. El estudiante asegura que la interfaz se actualice automáticamente al realizar cambios en la base de datos y que la aplicación mantenga un comportamiento estable. Esta semana refuerza los conocimientos de manejo de datos locales y prepara al estudiante para aplicar estilos y diseño avanzado en semanas posteriores.

| **Tema Principal** | **Conceptos Clave** | **Paquetes y Estructura** |
| --- | --- | --- |
| **Bases de Datos Locales** | **SQLite:** Bases de datos relacionales, ideal para datos **estructurados** (tareas, inventarios, notas). Soporta consultas SQL complejas. | Paquete **sqflite**, path\_provider. |
| **Operaciones CRUD** | Implementación de Crear, Leer, Actualizar, y Eliminar registros mediante sentencias SQL (o helpers del paquete). | **Database db;**, db.insert(...), db.query(...), db.update(...), db.delete(...). |
| **Modelo de Datos** | Es necesario definir un **Modelo (Clase Dart)** que mapee cada tabla de la base de datos para facilitar el manejo de los objetos en la aplicación. | Map<String, dynamic> toMap(), factory Clase.fromMap(Map<String, dynamic> map). |

**Bases de datos locales con SQLite**

**Foco Adicional:** **Mapeo de Objetos y la Estructura de la Base de Datos.**

* **SQL en Dart:** Se insiste en que el desarrollador debe estar cómodo con las consultas SQL básicas (CREATE TABLE, SELECT \* FROM, WHERE, UPDATE, DELETE).
* **Mapeo (De/A Map):** La parte más importante es el mapeo de objetos de Dart a formatos compatibles con SQLite (ej. Map<String, dynamic>) y viceversa.

Dart

// Convertir objeto Dart a Map para SQLite

Map<String, dynamic> toMap() {

return {'id': id, 'title': title};

}

// Convertir Map de SQLite a objeto Dart

factory Note.fromMap(Map<String, dynamic> map) {

return Note(id: map['id'], title: map['title']);

}

* **Versiones de la BD:** En la inicialización, se debe manejar la versión de la base de datos (onUpgrade: (db, oldVersion, newVersion)), crucial al desplegar una actualización con nuevos campos o tablas.

# EJEMPLO:

// nota.dart

class Note {

final int? id;

final String title;

Note({this.id, required this.title});

// Mapea el objeto Dart a un Map (para escribir en la DB)

Map<String, dynamic> toMap() {

return {

'id': id,

'title': title,

};

}

// Crea un objeto Dart desde un Map (leído de la DB)

factory Note.fromMap(Map<String, dynamic> map) {

return Note(

id: map['id'] as int?,

title: map['title'] as String,

);

}

}

// database\_helper.dart

import 'package:sqflite/sqflite.dart';

import 'package:path/path.dart';

class DatabaseHelper {

static Database? \_database;

static const String tableName = 'notes';

Future<Database> get database async {

if (\_database != null) return \_database!;

\_database = await \_initDB();

return \_database!;

}

\_initDB() async {

String path = join(await getDatabasesPath(), 'notes\_database.db');

// Abre la base de datos y crea la tabla si no existe

return await openDatabase(

path,

version: 1,

onCreate: (db, version) async {

await db.execute(

'''

CREATE TABLE $tableName(

id INTEGER PRIMARY KEY AUTOINCREMENT,

title TEXT

)

''',

);

},

);

}

// Operación CRUD: Insertar (Create)

Future<int> insertNote(Note note) async {

final db = await database;

return await db.insert(tableName, note.toMap(), conflictAlgorithm: ConflictAlgorithm.replace);

}

// Operación CRUD: Leer (Read)

Future<List<Note>> getNotes() async {

final db = await database;

final List<Map<String, dynamic>> maps = await db.query(tableName);

// Convierte List<Map> a List<Note>

return List.generate(maps.length, (i) => Note.fromMap(maps[i]));

}

}

# Semana 13: Diseño avanzado y temas

**Introducción al tema**

Durante la semana 13, el estudiante se enfoca en mejorar el **diseño visual** de la aplicación mediante la personalización de temas y estilos avanzados. Se comprende que una interfaz atractiva y consistente es fundamental para ofrecer una experiencia de usuario profesional, y que Flutter permite aplicar estilos globales que impactan a toda la aplicación de manera uniforme.

El objetivo principal de esta semana es que el estudiante aprenda a aplicar temas, modificar colores, tipografías, estilos de botones y otros elementos visuales para que la aplicación tenga una apariencia estética y coherente.

**Uso de ThemeData y estilos globales**

Se estudia el uso de **ThemeData** para definir un tema global dentro de la aplicación, que incluya colores primarios y secundarios, tipografía, estilos de botones, AppBar, texto y contenedores. Se enseña cómo alternar entre temas claros y oscuros, cómo aplicar estilos de manera consistente y cómo personalizar widgets específicos sin afectar el tema general. Se explica la importancia de mantener la coherencia visual para mejorar la experiencia del usuario y la profesionalización de la app.

**Creación de la práctica de diseño avanzado**

Durante la práctica, el estudiante desarrolla una aplicación básica aplicando un tema personalizado completo. Se modifican colores, tipografías, tamaños de texto, estilos de botones y AppBar, y se revisa la disposición de los elementos para lograr un diseño armónico y funcional. También se implementan iconos personalizados y se ajustan márgenes y paddings para mejorar la estética general. Esta actividad permite consolidar los conocimientos sobre diseño y mostrar cómo los estilos impactan la interacción del usuario.

**Ejecución y validación de la práctica**

Finalmente, se ejecuta la aplicación en un emulador o dispositivo físico para verificar que los temas y estilos se apliquen correctamente en todas las pantallas, que la interfaz sea visualmente atractiva y consistente, y que los cambios se reflejen de manera inmediata. El estudiante confirma que el diseño aplicado mejora la experiencia de usuario y sienta las bases para integrar animaciones, transiciones y funcionalidades más avanzadas en semanas posteriores.

| **Tema Principal** | **Conceptos Clave** | **Personalización y Widgets** |
| --- | --- | --- |
| **ThemeData Global** | Permite definir un conjunto de estilos (colores, fuentes, formas) que se aplicarán de forma consistente a todos los widgets de Material de la aplicación. | **ThemeData(...)** dentro de MaterialApp. |
| **Temas Claros/Oscuros** | Definir dos ThemeData y usar un StatefulWidget (a menudo con SharedPreferences) para permitir al usuario alternar entre modos. | theme: ThemeData.light(), darkTheme: ThemeData.dark(). |
| **Estilos Específicos** | Personalización de widgets de alto nivel (AppBar, Botones, FloatingActionButton) utilizando AppBarTheme, ButtonTheme, etc. Uso de Padding y Margin para el espaciado correcto. | primaryColor, fontFamily, padding y margin. |

**Diseño avanzado y temas**

**Foco Adicional:** **Diseño Adaptativo (MediaQuery) y Coherencia Visual.**

* **Coherencia Material Design:** Se introduce el concepto de que ThemeData ayuda a la aplicación a adherirse a los lineamientos de **Material Design** o **Cupertino** (para iOS).
* **Diseño Adaptativo (MediaQuery):** El diseño debe ser adaptable a diferentes tamaños y orientaciones. Se utiliza MediaQuery.of(context).size.width para tomar decisiones de diseño basadas en el tamaño real de la pantalla (ej. cambiar de Column a Row en pantallas anchas).
* **TextStyle Global:** El tema se utiliza para definir estilos de texto jerárquicos (textTheme.headlineSmall, textTheme.bodyMedium) y evitar la repetición de propiedades de fuente y tamaño en cada widget Text.

# EJEMPLO:

// Tema de la aplicación

final ThemeData lightTheme = ThemeData(

brightness: Brightness.light,

primarySwatch: Colors.teal,

appBarTheme: const AppBarTheme(

backgroundColor: Colors.teal,

titleTextStyle: TextStyle(color: Colors.white, fontSize: 20),

),

floatingActionButtonTheme: const FloatingActionButtonThemeData(

backgroundColor: Colors.pink,

),

);

final ThemeData darkTheme = ThemeData(

brightness: Brightness.dark,

primarySwatch: Colors.indigo,

appBarTheme: const AppBarTheme(

backgroundColor: Colors.black,

titleTextStyle: TextStyle(color: Colors.white, fontSize: 20),

),

);

class TemaDemo extends StatefulWidget {

const TemaDemo({super.key});

@override

State<TemaDemo> createState() => \_TemaDemoState();

}

class \_TemaDemoState extends State<TemaDemo> {

bool \_isDark = false;

void \_toggleTheme() {

setState(() {

\_isDark = !\_isDark;

});

}

@override

Widget build(BuildContext context) {

// Aplica el tema basado en el estado

return MaterialApp(

theme: lightTheme,

darkTheme: darkTheme,

themeMode: \_isDark ? ThemeMode.dark : ThemeMode.light, // Controla qué tema usar

home: Scaffold(

appBar: AppBar(

title: const Text('Semana 13: Temas'),

),

body: Center(

child: ElevatedButton(

onPressed: \_toggleTheme,

child: Text(\_isDark ? 'Cambiar a Claro' : 'Cambiar a Oscuro'),

),

),

floatingActionButton: FloatingActionButton(

onPressed: () {},

child: const Icon(Icons.palette),

),

),

);

}

}

# Semana 14: Animaciones en Flutter

**Introducción al tema**

Durante la semana 14, el estudiante se enfoca en aprender a implementar **animaciones y transiciones** en Flutter para mejorar la experiencia visual y la interacción del usuario. Se comprende que las animaciones permiten que la interfaz sea más dinámica, atractiva y fácil de entender, facilitando la navegación y la retroalimentación visual.

El objetivo de esta semana es que el estudiante pueda aplicar animaciones simples y complejas para dar vida a los widgets y crear una experiencia de usuario más profesional.

**Tipos de animaciones en Flutter**

Se estudian diferentes tipos de animaciones, incluyendo **animaciones implícitas** y **animaciones controladas**. Las animaciones implícitas permiten modificar propiedades de un widget de forma sencilla, como tamaño, color u opacidad, usando widgets como **AnimatedContainer**, **AnimatedOpacity** y **AnimatedAlign**. Las animaciones controladas, por su parte, ofrecen mayor precisión y control mediante **AnimationController** y **Tween**, permitiendo crear transiciones más complejas y sincronizadas.

**Creación de animaciones prácticas**

Durante la práctica, el estudiante desarrolla animaciones para botones, contenedores y textos. Por ejemplo, se implementa un contenedor que cambia de color y tamaño al ser presionado, y un texto que se desplaza o aparece suavemente con transiciones. Además, se combinan animaciones con **StatefulWidgets** para que la interfaz responda a la interacción del usuario en tiempo real.

**Transiciones entre pantallas**

Se estudia cómo aplicar animaciones al cambiar de pantalla usando **PageRouteBuilder** y transiciones predefinidas como **fade**, **slide** o **scale**. Esto permite que la navegación entre pantallas sea más fluida y visualmente agradable, mejorando la percepción de calidad de la aplicación.

**Ejecución y validación**

Finalmente, se ejecuta la aplicación en un emulador o dispositivo físico para verificar que las animaciones se reproduzcan correctamente, que no afecten el rendimiento y que respondan a la interacción del usuario. El estudiante valida que las transiciones entre pantallas sean suaves y que la experiencia visual sea coherente y profesional.

| **Tema Principal** | **Conceptos Clave** | **Widgets y Clases** |
| --- | --- | --- |
| **Conceptos Fundamentales** | Las animaciones son la base de las interfaces fluidas. Flutter utiliza un sistema de alto rendimiento que se ejecuta a **60 cuadros por segundo (fps)**. | **Ticker** (mecanismo de sincronización), **AnimationController** (gestiona el tiempo/duración de la animación), **Tween** (define el rango de valores). |
| **Animaciones Implícitas** | Widgets que se animan automáticamente cuando sus propiedades cambian. Son la forma más sencilla de añadir movimiento sin controladores manuales. Ideales para cambios de color, tamaño, o posición simple. | **AnimatedContainer**, **AnimatedOpacity**, **AnimatedPositioned**. |
| **Animaciones Explícitas** | Requieren un control manual del ciclo de vida y la duración (usando AnimationController). Son necesarias para animaciones complejas, encadenadas o repetitivas. | **AnimationController**, **SingleTickerProviderStateMixin** (para el StatefulWidget), **FadeTransition**, **RotationTransition**. |
| **Práctica Avanzada** | Creación de una animación de carga personalizada (spinner) o una transición de pantalla que modifique el tamaño y la opacidad simultáneamente, usando Tween para mapear los valores entre el inicio y el final. | controller.forward(), controller.repeat(). |

**Animaciones en Flutter**

**Foco Adicional:** **Curvas de Animación y Composición de Animaciones.**

* **Curvas (Curves):** Se explica que el Tween define los puntos de inicio y fin, pero la **Curva** define la "sensación" o velocidad de la animación (Curves.easeIn, Curves.bounceOut).
* **Composición de Animaciones:** Se muestra cómo usar el mismo AnimationController con múltiples objetos Tween para animar simultáneamente el tamaño, la opacidad y el color de un widget (ej. SizeTransition y FadeTransition usando el mismo Animation).
* **Hero Widget:** Un widget de animación muy popular que realiza una transición visual fluida entre la misma imagen/elemento en dos pantallas diferentes (navegación por detalles).

# EJEMPLO:

class AnimacionImplicitaDemo extends StatefulWidget {

const AnimacionImplicitaDemo({super.key});

@override

State<AnimacionImplicitaDemo> createState() => \_AnimacionImplicitaDemoState();

}

class \_AnimacionImplicitaDemoState extends State<AnimacionImplicitaDemo> {

bool \_grande = false;

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(title: const Text('Semana 14: AnimatedContainer')),

body: Center(

// El AnimatedContainer maneja la transición automáticamente

child: AnimatedContainer(

duration: const Duration(milliseconds: 500), // Duración de la animación

curve: Curves.easeInOut, // Curva de velocidad

width: \_grande ? 200.0 : 100.0,

height: \_grande ? 200.0 : 100.0,

decoration: BoxDecoration(

color: \_grande ? Colors.deepPurple : Colors.cyan,

borderRadius: BorderRadius.circular(\_grande ? 50.0 : 5.0),

),

child: Center(

child: Text(

\_grande ? 'Grande' : 'Pequeño',

style: const TextStyle(color: Colors.white),

),

),

),

),

floatingActionButton: FloatingActionButton(

onPressed: () {

setState(() {

\_grande = !\_grande; // Cambia el estado para disparar la animación

});

},

child: const Icon(Icons.play\_arrow),

),

);

}

}

# Semana 15: Integración con Firebase – Autenticación

**Introducción al tema**

Durante la semana 15, el estudiante aprende a integrar **librerías externas y plugins** en Flutter para ampliar la funcionalidad de la aplicación. Se comprende que el ecosistema de Flutter ofrece paquetes que permiten agregar funcionalidades como mapas, cámaras, notificaciones push, reproducción de videos, gráficos y más, sin necesidad de desarrollar todo desde cero.

El objetivo es que el estudiante pueda identificar, instalar y utilizar paquetes externos para mejorar la funcionalidad y eficiencia de sus aplicaciones.

**Instalación y configuración de paquetes**

Se estudia cómo buscar paquetes en **pub.dev**, cómo añadirlos en el archivo **pubspec.yaml** y cómo ejecutar los comandos necesarios para instalar y actualizar dependencias. Se explica la importancia de leer la documentación de cada paquete para comprender su funcionamiento, dependencias y compatibilidad con la versión de Flutter utilizada.

**Uso práctico de plugins**

Durante la práctica, el estudiante integra al menos un plugin externo en su proyecto, como la reproducción de imágenes desde la galería, mapas interactivos, generación de códigos QR o notificaciones locales. Se aprende a inicializar la librería, configurar permisos si es necesario y utilizar los métodos y widgets que proporciona para interactuar con la aplicación.

**Validación de la integración**

Se ejecuta la aplicación en un emulador o dispositivo físico para comprobar que el plugin funcione correctamente, que no genere errores y que la interacción con otros elementos de la aplicación sea estable. El estudiante valida la correcta instalación, configuración y uso del plugin, comprendiendo cómo estas herramientas amplían las capacidades de Flutter sin complicaciones adicionales.

| **Tema Principal** | **Conceptos Clave** | **Paquetes y Métodos de Firebase** |
| --- | --- | --- |
| **Introducción a Firebase** | Plataforma de Google que proporciona servicios de backend (bases de datos, almacenamiento, autenticación) listos para usar, eliminando la necesidad de construir un servidor propio. | **Firebase Console**, Configuración del proyecto (GoogleServices-Info.plist/google-services.json). |
| **Configuración de Autenticación** | Habilitación de métodos de inicio de sesión (Correo/Contraseña, Google, Anónimo) en la consola de Firebase. | Paquete **firebase\_core**, Paquete **firebase\_auth**. |
| **Implementación de Flujos** | Creación de formularios que interactúan con Firebase para: **Registro** de nuevos usuarios, **Inicio de Sesión** y **Cierre de Sesión**. Manejo de excepciones (ej. "contraseña incorrecta" o "correo ya en uso"). | FirebaseAuth.instance.createUserWithEmailAndPassword(), signInWithEmailAndPassword(), signOut(). |
| **Manejo del Estado de Autenticación** | Uso de **StreamBuilder** para escuchar cambios en el estado del usuario (conectado vs. desconectado). Esto permite redirigir automáticamente al usuario a la pantalla de inicio o a la pantalla de inicio de sesión. | **StreamBuilder** con FirebaseAuth.instance.authStateChanges(). |

**Integración con Firebase – Autenticación**

**Foco Adicional:** **ID Token y Seguridad.**

* **Gestión del Usuario Actual:** El objeto User obtenido de Firebase contiene el UID, que es el identificador único del usuario. Este UID es fundamental para la seguridad y para saber qué datos le pertenecen en Firestore.
* **Token de ID:** Se explica que, una vez autenticado, Firebase emite un **ID Token** que debe usarse para autenticar las solicitudes a cualquier otra API propia, asegurando que solo usuarios verificados accedan a recursos protegidos.
* **Recuperación de Contraseña:** Implementación del flujo de "Olvidé mi Contraseña" utilizando FirebaseAuth.instance.sendPasswordResetEmail().

# EJEMPLO:

// Nota: Este código es conceptual y requiere la inicialización de Firebase

// en el main() y las dependencias (firebase\_core, firebase\_auth).

import 'package:firebase\_auth/firebase\_auth.dart';

class AuthServiceDemo extends StatelessWidget {

const AuthServiceDemo({super.key});

@override

Widget build(BuildContext context) {

// 1. StreamBuilder escucha los cambios de estado del usuario

return StreamBuilder<User?>(

stream: FirebaseAuth.instance.authStateChanges(), // Escucha si el usuario cambia

builder: (context, snapshot) {

// 2. Si hay datos, el usuario está logueado

if (snapshot.hasData) {

return PantallaLogueada(user: snapshot.data!); // Muestra la pantalla principal

} else {

// 3. Si no hay datos, muestra la pantalla de Login

return const PantallaLogin();

}

},

);

}

}

class PantallaLogin extends StatelessWidget {

const PantallaLogin({super.key});

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(title: const Text('Semana 15: Login')),

body: Center(

child: ElevatedButton(

// Inicia sesión con credenciales de prueba

onPressed: () async {

try {

await FirebaseAuth.instance.signInWithEmailAndPassword(

email: 'test@example.com',

password: 'password123',

);

} catch (e) {

print('Error de login: $e');

}

},

child: const Text('Iniciar Sesión (Demo)'),

),

),

);

}

}

class PantallaLogueada extends StatelessWidget {

final User user;

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

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(title: const Text('Bienvenido')),

body: Center(

child: Column(

mainAxisAlignment: MainAxisAlignment.center,

children: [

Text('Usuario UID: ${user.uid}'),

ElevatedButton(

// 4. Cierra la sesión

onPressed: () async {

await FirebaseAuth.instance.signOut();

},

child: const Text('Cerrar Sesión'),

),

],

),

),

);

}

}

# Semana 16: Integración con Firebase – Firestore

**Introducción al tema**

Durante la semana 16, el estudiante realiza una **revisión general de todos los temas aprendidos** a lo largo del curso, consolidando conocimientos sobre Flutter, widgets, navegación, manejo de estado, consumo de APIs, almacenamiento local, diseño, animaciones y librerías externas. Se comprende la importancia de organizar la información y evidencias de manera clara para documentar el aprendizaje.

El objetivo es que el estudiante compile todas las prácticas, revisiones y proyectos realizados durante el curso para presentarlas como carpeta de evidencias completa y coherente.

**Revisión de conceptos y prácticas**

El estudiante repasa cada tema, verificando que los proyectos creados funcionen correctamente y que todas las funcionalidades estén implementadas de manera eficiente. Se revisan prácticas como:

* Creación de interfaces con widgets básicos y avanzados.
* Manejo de listas y formularios.
* Navegación entre pantallas y manejo de estado.
* Consumo de APIs y almacenamiento de datos.
* Aplicación de temas, animaciones y librerías externas.

Se corrigen errores y se mejoran detalles de diseño y funcionalidad para asegurar que cada práctica sea clara y funcional.

**Organización de la carpeta de evidencias**

El estudiante recopila capturas de pantalla, explicaciones de cada práctica, fragmentos de código relevantes y descripciones de las actividades realizadas en cada semana. Se organizan los documentos por semana y por temática, asegurando que la carpeta muestre de manera estructurada todo el aprendizaje adquirido.

**Preparación y validación final**

Finalmente, el estudiante revisa que todos los archivos y documentos estén completos, que las explicaciones sean claras y que las capturas de pantalla muestren los resultados de las prácticas. Esta revisión final asegura que la carpeta de evidencias refleje de manera profesional todo el aprendizaje del curso y esté lista para presentarse como evidencia del dominio de Flutter y desarrollo de aplicaciones

| **Tema Principal** | **Conceptos Clave** | **Estructura y Métodos de Firestore** |
| --- | --- | --- |
| **Cloud Firestore (Base de Datos NoSQL)** | Base de datos basada en **Documentos** y **Colecciones**. Es ideal para datos jerárquicos y flexibles. Ofrece **Sincronización en Tiempo Real** de los datos entre dispositivos. | **Colecciones** (tablas), **Documentos** (filas/registros), **Campos** (columnas). |
| **Operaciones CRUD en Firestore** | Métodos para escribir, leer, actualizar y eliminar datos de los documentos. Es crucial el manejo de rutas de colecciones y documentos para acceder a la información correcta. | Paquete **cloud\_firestore**. |
| **Lectura en Tiempo Real** | Uso de **StreamBuilder** (similar a la autenticación) para suscribirse a los cambios en una colección. Si alguien en cualquier parte del mundo actualiza un documento, la UI se actualiza inmediatamente sin necesidad de recargar. | **FirebaseFirestore.instance.collection('ruta').snapshots()**. |
| **Práctica de Integración** | Implementar una aplicación de "Lista de Tareas" o "Chat Básico" que use Firestore para almacenar y sincronizar los mensajes o tareas entre múltiples usuarios, integrando los datos de Autenticación para identificar al autor. | collection.add(data), document.update(data). |

**Integración con Firebase – Firestore**

**Foco Adicional:** **Seguridad (Rules) e Indexación.**

* **Seguridad de Datos (Firebase Rules):** Se enfatiza que el código de la aplicación es cliente, por lo que **no se debe confiar en él**. La seguridad de los datos debe ser impuesta en la capa del servidor mediante **Firestore Security Rules**.
* // Ejemplo de regla de seguridad
* match /users/{userId} {
* allow read, write: if request.auth.uid == userId;
* }
* **Indexación:** Se explica que, al realizar consultas complejas (ej. where('status', '==', 'pending').orderBy('date')), Firestore puede requerir la creación de **Índices** manuales en la consola de Firebase para funcionar y optimizar la velocidad.
* **Transacciones (Concepto):** Operaciones que garantizan que múltiples escrituras de datos se completen por completo o fallen por completo, manteniendo la integridad de los datos (importante en sistemas de inventario o saldos).

# EJEMPLO:

// Nota: Este código es conceptual y requiere las dependencias (cloud\_firestore).

import 'package:cloud\_firestore/cloud\_firestore.dart';

class FirestoreDemo extends StatelessWidget {

const FirestoreDemo({super.key});

@override

Widget build(BuildContext context) {

// 1. Referencia a la colección (ejemplo: tareas)

final CollectionReference tareas =

FirebaseFirestore.instance.collection('tareas');

return Scaffold(

appBar: AppBar(title: const Text('Semana 16: Firestore Live')),

// 2. StreamBuilder escucha los 'snapshots' (cambios en tiempo real)

body: StreamBuilder<QuerySnapshot>(

stream: tareas.snapshots(),

builder: (context, snapshot) {

if (snapshot.hasError) {

return Center(child: Text('Algo salió mal: ${snapshot.error}'));

}

if (snapshot.connectionState == ConnectionState.waiting) {

return const Center(child: CircularProgressIndicator());

}

// 3. Datos disponibles: Los documentos (docs) de la colección

final List<DocumentSnapshot> documentos = snapshot.data!.docs;

return ListView.builder(

itemCount: documentos.length,

itemBuilder: (context, index) {

final data = documentos[index].data() as Map<String, dynamic>;

final String docId = documentos[index].id;

return ListTile(

title: Text(data['titulo'] ?? 'Sin Título'),

subtitle: Text('ID: $docId'),

trailing: IconButton(

icon: const Icon(Icons.delete),

// 4. Operación CRUD: Eliminar

onPressed: () => tareas.doc(docId).delete(),

),

onTap: () {

// Operación CRUD: Actualizar

tareas.doc(docId).update({'completado': true});

},

);

},

);

},

),

floatingActionButton: FloatingActionButton(

// 5. Operación CRUD: Crear (Agregar nuevo documento)

onPressed: () => tareas.add({'titulo': 'Nueva Tarea ${DateTime.now().second}', 'completado': false}),

child: const Icon(Icons.add),

),

);

}

}