MANUAL FRONTEND

FLUTTER & ANDROID STUDIO

FLUTTER

INTRODUCCIÓN

Flutter es un framework de desarrollo de aplicaciones de código abierto creado por Google, que permite construir aplicaciones nativas para múltiples plataformas usando un solo código base. Con Flutter, se pueden desarrollar aplicaciones para iOS, Android, web y escritorio (Windows, macOS y Linux) sin tener que escribir código diferente para cada plataforma.

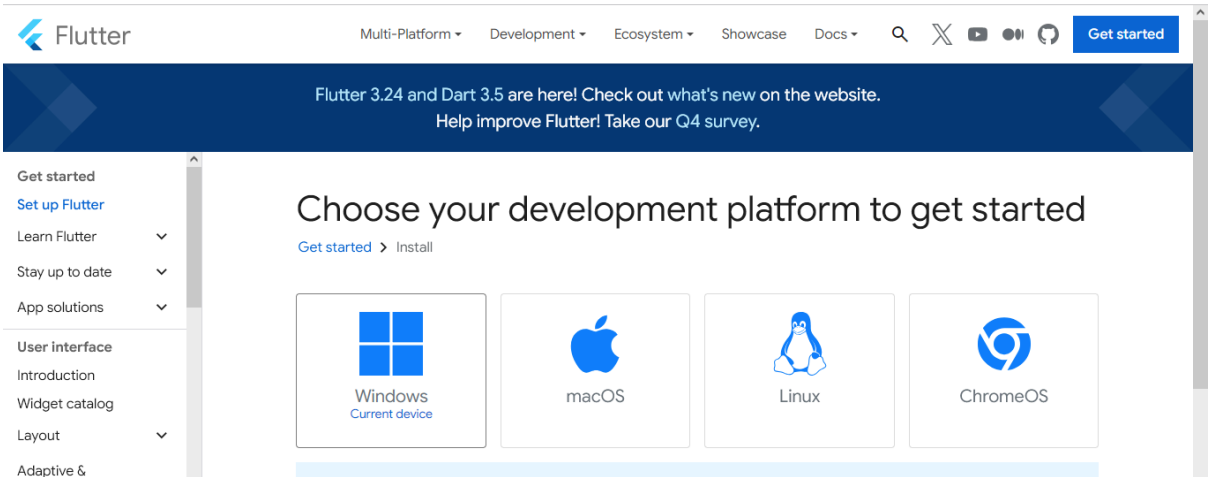
El framework usa un lenguaje de programación llamado **Dart**, también desarrollado por Google. En lugar de usar los componentes de interfaz de usuario nativos de cada plataforma (como los botones de Android o iOS), Flutter dibuja sus propios componentes en una capa gráfica. Esto se hace usando **widgets**, los bloques básicos que Flutter usa para construir interfaces. Todo, desde un texto hasta una estructura compleja, es un widget en Flutter, lo que permite una gran flexibilidad para personalizar las aplicaciones.

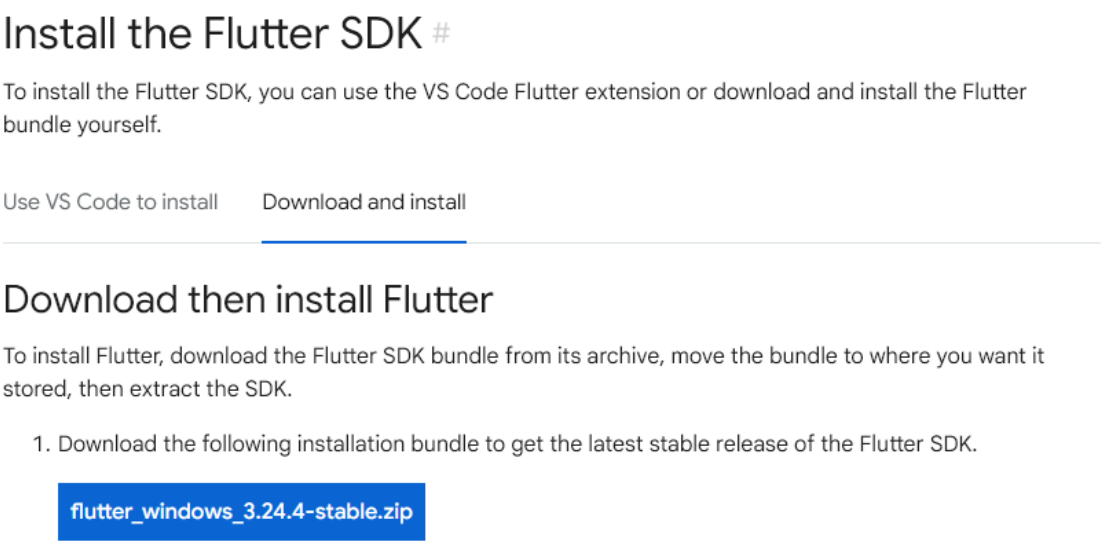
Existe un repositorio oficial de Dart y Flutter llamado pub.dev , donde se publican librerías para usar en proyectos.

INSTALACIÓN

Paso 1: Descarga de Flutter

- Ir a la página oficial de Flutter: flutter.dev y seleccionar **Get Started**.
- Escoger el sistema operativo que está usando (Windows, macOS o Linux) y descarga el SDK de Flutter.





Paso 2: Configurar Flutter

1. **Crear en el Disco C** una carpeta llamada src
2. **Extraer el archivo descargado** en `C:\src\`
3. **Agrega Flutter al PATH** para poder ejecutarlo desde cualquier parte:
    - En **Windows**:
        - Ir a las **Variables de Entorno** en la configuración del sistema.
        - En **Variables de Usuario**, seleccionar `PATH` y editar la variable.
        - Agrega la ruta a la carpeta `flutter/bin` (ej. `C:\src\flutter\bin`).
    - En **macOS o Linux**:
        - Abre el archivo `.bashrc` o `.zshrc` (dependiendo de tu shell) y agregar:

```
export PATH="$PATH:[ruta/a/flutter]/bin"
```

        - Guarda el archivo y ejecuta `source .bashrc` o `source .zshrc` para actualizar el PATH.

Paso 3: Verificar la instalación

1. Abrir una terminal o consola y ejecutar:

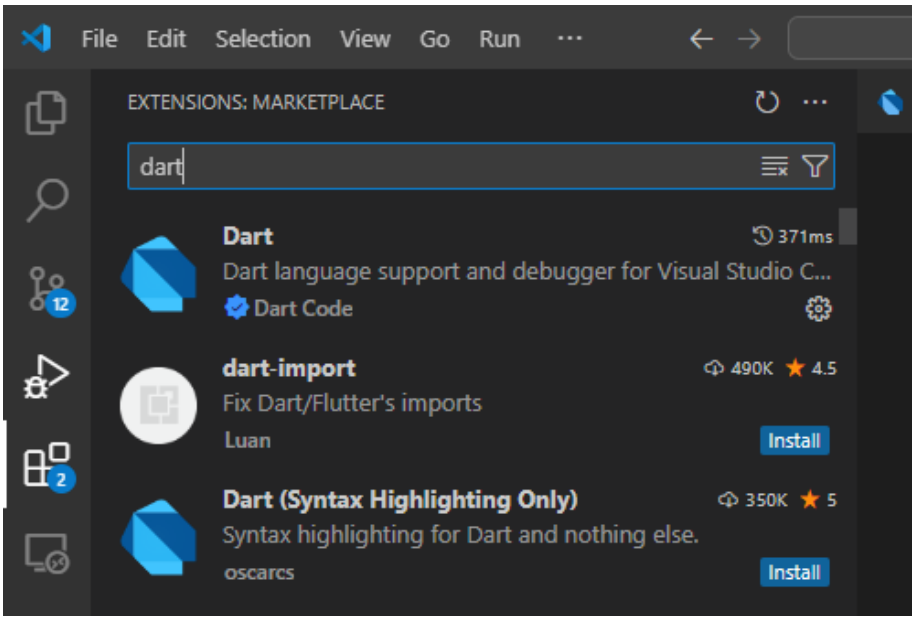
```
flutter doctor
```

   Este comando verificará si tiene las herramientas necesarias para correr Flutter, como el SDK de Android, el emulador y las dependencias adicionales. Mostrará una lista de lo que está instalado y lo que falta para completar la configuración.

```
Doctor summary (to see all details, run flutter doctor -v):
[√] Flutter (Channel stable, 3.7.11, on Microsoft Windows [Versión 10.0.19045.5011], locale es-EC)
[X] Windows Version (Unable to confirm if installed Windows version is 10 or greater)
Checking Android licenses is taking an unexpectedly long time...[√] Android toolchain - develop for Android devices (Android SDK version 34.0.0
)
[√] Chrome - develop for the web
[√] Visual Studio - develop for Windows (Visual Studio Community 2022 17.1.3)
[!] Android Studio (version 2021.1)
    X Unable to determine bundled Java version.
[!] Android Studio (version 2024.1)
    X Unable to find bundled Java version.
[√] VS Code (version 1.95.2)
[√] Connected device (3 available)
[√] HTTP Host Availability
```

Paso 4: Instalar VS Code

1. Descarga e instala **VS Code** desde code.visualstudio.com.
2. En la sección de **Extensiones**, instala los plugins de **Flutter** y **Dart**.



CREAR PROYECTO

1. Abrir una consola para ejecutar el siguiente comando:

```
flutter create nombre_proyecto
```

```
PC@DESKTOP-11SJSE3 MINGW64 ~/Desktop/repositorios
$ flutter create proyecto1
```

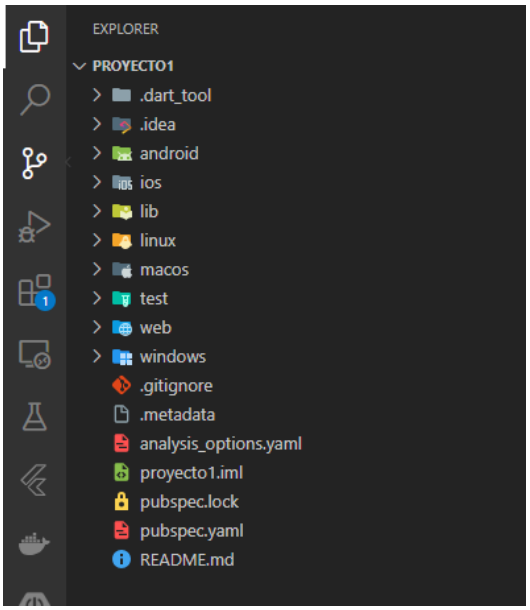
Esto generará una estructura básica del proyecto

```
Creating project proyecto1...
Running "flutter pub get" in proyecto1...
Resolving dependencies in proyecto1...
+ async 2.10.0 (2.12.0 available)
+ boolean_selector 2.1.1 (2.1.2 available)
+ characters 1.2.1 (1.3.1 available)
+ clock 1.1.1 (1.1.2 available)
+ collection 1.17.0 (1.19.1 available)
+ cupertino_icons 1.0.6 (1.0.8 available)
+ fake_async 1.3.1 (1.3.2 available)
+ flutter 0.0.0 from sdk flutter
+ flutter_lints 2.0.3 (5.0.0 available)
+ flutter_test 0.0.0 from sdk flutter
+ js 0.6.5 (0.7.1 available)
+ lints 2.0.1 (5.1.0 available)
+ matcher 0.12.13 (0.12.17 available)
+ material_color_utilities 0.2.0 (0.12.0 available)
+ meta 1.8.0 (1.16.0 available)
+ path 1.8.2 (1.9.1 available)
+ sky_engine 0.0.99 from sdk flutter
+ source_span 1.9.1 (1.10.0 available)
+ stack_trace 1.11.0 (1.12.0 available)
+ stream_channel 2.1.1 (2.1.2 available)
+ string_scanner 1.2.0 (1.4.0 available)
+ term_glyph 1.2.1
+ test_api 0.4.16 (0.7.3 available)
+ vector_math 2.1.4
Changed 24 dependencies in proyecto1!
Wrote 127 files.

All done!
You can find general documentation for Flutter at: https://docs.flutter.dev/
Detailed API documentation is available at: https://api.flutter.dev/
If you prefer video documentation, consider: https://www.youtube.com/c/flutterde
v

In order to run your application, type:

  $ cd proyecto1
  $ flutter run

Your application code is in proyecto1\lib\main.dart.
```

2. Abrir el proyecto con VS Code

```
PC@DESKTOP-11SJSE3 MINGW64 ~/Desktop/repositorios
$ cd proyecto1

PC@DESKTOP-11SJSE3 MINGW64 ~/Desktop/repositorios/proyecto1
$ code .
```

Y se visualizarán todos los archivos creados automáticamente



ESTRUCTURA DEL PROYECTO

Cuando se crea un proyecto en Flutter, se genera una estructura de carpetas y archivos que organizan el código, las configuraciones y los recursos necesarios. A continuación, se explica cada parte de la estructura:

Carpetas Principales

`android/`

- Contiene todo el código nativo relacionado con la plataforma Android.
- Usado cuando necesitas integrar características específicas de Android o configurar el proyecto a nivel nativo.
- Archivos importantes:
    - **`build.gradle`** **(en `android/` y `android/app/`)**: Contienen configuraciones para la compilación de la aplicación en Android.
    - **`AndroidManifest.xml`**: Configura permisos y metadatos para la aplicación en Android.
    - **`MainActivity.kt`** **o** **`MainActivity.java`**: Punto de entrada para la aplicación en Android.

`ios/`

- Contiene el código nativo para iOS.
- Usado para configuraciones específicas de iOS o integraciones nativas.
- Archivos importantes:
    - **`Info.plist`**: Archivo de configuración para la aplicación en iOS (metadatos como nombre de la app, iconos, permisos, etc.).
    - **`AppDelegate.swift`** **o** **`AppDelegate.m`**: Punto de entrada para la aplicación en iOS.

`lib/`

- Es la carpeta principal donde escribirás el código de tu aplicación en Dart.
- Contiene el archivo principal del proyecto:
    - **`main.dart`**: Es el punto de entrada de tu aplicación Flutter. Aquí defines la estructura básica y lógica inicial.
- Dentro de esta carpeta, se puede crear subcarpetas para organizar el código, como:
    - **`screens/`**: Para las pantallas de la aplicación.
    - **`widgets/`**: Para los widgets personalizados.
    - **`models/`**: Para las clases que representen datos.
    - **`services/`**: Para la lógica de negocio o comunicación con APIs.

`test/`

- Contiene las pruebas unitarias y funcionales de tu aplicación.
- Aquí puedes escribir pruebas para asegurar que tu aplicación funciona correctamente.
- Archivos terminan generalmente en `_test.dart`.

`web/`

Esta carpeta contiene los archivos necesarios para ejecutar tu aplicación Flutter como una aplicación web (HTML, CSS, y JavaScript). Solo aparece si habilitas Flutter para web.

- **`index.html`**:
  - Es el archivo HTML principal que sirve como punto de entrada para la aplicación web.
  - Contiene un contenedor `<div>` donde Flutter renderiza la interfaz de usuario.
  - Puedes personalizarlo para incluir metadatos, scripts adicionales o enlaces a hojas de estilo.
- **`favicon.png`**:
  - Es el ícono que aparece en la pestaña del navegador al abrir tu aplicación web.
  - Puedes reemplazarlo con tu propio ícono.
- **`manifest.json`**:
  - Define cómo la aplicación se comporta cuando se instala como una Progressive Web App (PWA).
  - Especifica nombre, descripción, íconos y temas de la aplicación.
- **`web.yaml`** **(opcional)**:
  - Archivo de configuración adicional para personalizar aspectos avanzados de la aplicación web.

`windows/`

Esta carpeta contiene los archivos necesarios para construir una aplicación Flutter que se ejecute de forma nativa en Windows.

- **`CMakeLists.txt`**:
  - Un archivo de configuración para el sistema de construcción CMake.
  - Define cómo compilar y vincular el código de tu aplicación para Windows.
- **`runner/`**:
  - Contiene el código específico del "corredor" de la aplicación (el contenedor que ejecuta tu aplicación Flutter en Windows).
  - Archivos importantes:
    - **`main.cpp`**: Es el punto de entrada para la aplicación en Windows.
    - **`flutter_window.cpp`**: Configura la ventana principal de la aplicación en Windows, como tamaño, título y opciones.

`linux/`

Esta carpeta contiene los archivos necesarios para ejecutar tu aplicación Flutter en sistemas Linux.

- `CMakeLists.txt`:
  - Similar al de Windows, este archivo configura el sistema de construcción para compilar la aplicación en Linux.
- `runner/`:
  - Archivos específicos para integrar tu aplicación con el sistema Linux.
  - Archivos importantes:
    - `main.cc`: Es el punto de entrada principal de la aplicación en Linux.
    - `flutter_window.cc`: Configura la ventana principal y su interacción con el entorno de Linux.

`macos/`

Esta carpeta contiene los archivos necesarios para construir una aplicación Flutter nativa para macOS.

- `AppDelegate.swift`:
  - Archivo Swift que sirve como punto de entrada nativo para la aplicación en macOS.
  - Configura el entorno de la aplicación antes de iniciar Flutter.
- `MainFlutterWindow.swift`:
  - Configura la ventana principal de la aplicación, como el tamaño inicial y el título.
- `Runner.xcworkspace` **y** `Runner.xcodeproj`:
  - Archivos de configuración de Xcode para compilar y gestionar la aplicación en macOS.
- `Info.plist`:
  - Archivo de configuración que define metadatos sobre la aplicación, como permisos y configuración del sistema.

Archivos Principales

`pubspec.yaml`

Archivo de configuración del proyecto en Flutter.

Define:

- **Dependencias**: Librerías que tu proyecto necesita para funcionar.
- **Assets**: Archivos como imágenes, fuentes o configuraciones personalizadas.

```yaml
name: proyecto1

description: Un proyecto Flutter simple.

version: 1.0.0+1


environment:

  sdk: ">=3.0.0 <4.0.0" # Versión mínima y máxima de Dart soportada.


dependencies:

  flutter:

      sdk: flutter


  # Librerías comunes

  cupertino_icons: ^1.0.2  # Iconos estilo iOS

  http: ^0.15.0          # Para realizar solicitudes HTTP

  provider: ^6.0.0       # Gestión de estado simple


dev_dependencies:

  flutter_test:

      sdk: flutter


flutter:

  # Configuración para assets

  assets:
```

```
    - assets/images/logo.png  # Imágenes
    - assets/data/config.json # Archivos de configuración


# Configuración para fuentes personalizadas
fonts:
    - family: Roboto
    fonts:
    - asset: assets/fonts/Roboto-Regular.ttf
    - asset: assets/fonts/Roboto-Bold.ttf
    weight: 700
```

Explicación de las secciones:

1. **`name` y `description`:**
    - Define el nombre y la descripción del proyecto.
2. **`version`:**
    - Especifica la versión de tu aplicación.
    - El formato es `X.Y.Z+build`, donde `X.Y.Z` es la versión y `build` es el número de compilación. Este parámetro se deberá cambiar cada que se haga alguna actualización en la aplicación.
3. **`environment`:**
    - Define las versiones de Dart compatibles con el proyecto.
4. **`dependencies`:**
    - Lista las dependencias (librerías) necesarias para la aplicación.
    - Incluye:
        - **`flutter`:** La dependencia base de Flutter.
        - **`cupertino_icons`:** Librería común para iconos estilo iOS.
        - Otras dependencias como `http` y `provider`.
5. **`dev_dependencies`:**
    - Dependencias sólo necesarias durante el desarrollo, como herramientas de prueba.
6. **`flutter`:**
    - **`assets`:** Define los recursos que usarás en tu proyecto, como imágenes, fuentes y archivos JSON.
    - **`fonts`:** Permite agregar fuentes personalizadas.

ESTRUCTURA TÍPICA DE CÓDIGO

Dentro de la carpeta `lib/`, el archivo principal es **`main.dart`**, que contiene el punto de entrada de la aplicación. Sin embargo, para proyectos más grandes, es una buena práctica organizar el código en subcarpetas con roles específicos. Por ejemplo la siguiente estructura:

```
lib/
├── main.dart           # Punto de entrada de la aplicación.
├── screens/            # Pantallas principales de la aplicación.
|   ├── home_screen.dart # Pantalla de inicio.
|   ├── login_screen.dart # Pantalla de inicio de sesión.
├── widgets/            # Widgets reutilizables y personalizados.
|   ├── custom_button.dart # Botón reutilizable.
|   ├── header.dart     # Encabezado común entre pantallas.
├── models/             # Modelos de datos y estructuras.
|   ├── user.dart       # Modelo de usuario.
|   ├── product.dart    # Modelo de producto.
├── services/           # Lógica de negocio y comunicación con APIs.
|   ├── api_service.dart # Servicio para manejar solicitudes HTTP.
|   ├── auth_service.dart # Servicio para autenticación.
├── utils/              # Utilidades generales, temas y constantes.
|   ├── theme.dart      # Configuración del tema (colores, fuentes).
|   ├── constants.dart # Constantes globales.
|   ├── helpers.dart    # Funciones útiles.
└── assets/             # Recursos de la app, como imágenes y fuentes
(se referencia en `pubspec.yaml`).
```

EJEMPLO DE APLICACIÓN

A continuación se muestra el desarrollo de una aplicación simple que muestra los medios de pago disponibles en la empresa.

La estructura será:

```
lib/
├── main.dart
├── models/
│   └── medio_pago.dart
├── services/
│   └── api_service.dart
├── screens/
│   └── medios_pago_screen.dart
```

Modelo de Datos: `medio_pago.dart`

Este modelo representa cada medio de pago obtenido desde la API.

```dart
class MedioPago {
  final int idMedioPagoTicket;
  final String medioPago;
  final bool anulado;
  final bool cargarDocDig;

  MedioPago({
      required this.idMedioPagoTicket,
      required this.medioPago,
      required this.anulado,
      required this.cargarDocDig,
  });

  // Método para deserializar desde JSON
  factory MedioPago.fromJson(Map<String, dynamic> json) {
      return MedioPago(
      idMedioPagoTicket: json['idMedioPagoTicket'],
      medioPago: json['medioPago'],
      anulado: json['anulado'],
```

```
      cargarDocDig: json['cargarDocDig'],
      );
  }
}
```

Servicio API: api_service.dart

Este servicio maneja las solicitudes HTTP para obtener los medios de pago.

```
import 'dart:convert';
import 'package:http/http.dart' as http;
import '../models/medio_pago.dart';

class ApiService {
  static const String baseUrl = 'https://saitelapp.ec:15000';

  Future<List<MedioPago>> fetchMediosPago() async {
      final url =
Uri.parse('$baseUrl/ticketsweb/ticket/pago/medios');

      try {
      final response = await http.get(url);

      if (response.statusCode == 200) {
      // Decodificar la respuesta JSON
      final List<dynamic> jsonData = json.decode(response.body);

      // Convertir JSON en una lista de objetos MedioPago
      return jsonData.map((item) =>
MedioPago.fromJson(item)).toList();
      } else {
      throw Exception('Error al obtener los medios de pago');
      }
      } catch (e) {
      throw Exception('Error de conexión: $e');
      }
  }
}
```

Pantalla: **medios_pago_screen.dart**
Esta pantalla muestra la lista de medios de pago obtenida desde la API.

```
import 'package:flutter/material.dart';
import '../services/api_service.dart';
import '../models/medio_pago.dart';

class MediosPagoScreen extends StatefulWidget {
  @override
  _MediosPagoScreenState createState() =>
_MediosPagoScreenState();
}

class _MediosPagoScreenState extends State<MediosPagoScreen> {
  late Future<List<MedioPago>> _mediosPagoFuture;

  @override
  void initState() {
      super.initState();
      _mediosPagoFuture = ApiService().fetchMediosPago();
  }

  @override
  Widget build(BuildContext context) {
      return Scaffold(
      appBar: AppBar(
      title: Text('Medios de Pago'),
      ),
      body: FutureBuilder<List<MedioPago>>(
      future: _mediosPagoFuture,
      builder: (context, snapshot) {
           if (snapshot.connectionState ==
ConnectionState.waiting) {
           return Center(child: CircularProgressIndicator());
           } else if (snapshot.hasError) {
           return Center(child: Text('Error:
${snapshot.error}'));
           } else if (!snapshot.hasData ||
snapshot.data!.isEmpty) {
```

```
            return Center(child: Text('No hay medios de pago
disponibles.'));
            } else {
            final mediosPago = snapshot.data!;
            return ListView.builder(
            itemCount: mediosPago.length,
            itemBuilder: (context, index) {
                  final medio = mediosPago[index];
                  return ListTile(
                  title: Text(medio.medioPago),
                  subtitle: Text(
                        'ID: ${medio.idMedioPagoTicket} - Cargar
Documento: ${medio.cargarDocDig ? "Sí" : "No"}'),
                  trailing: Icon(
                  medio.anulado ? Icons.cancel :
Icons.check_circle,
                  color: medio.anulado ? Colors.red : Colors.green,
                  ),
                  );
            },
            );
            }
      },
      ),
      );
  }
}
```

Archivo principal : main.dart

Conecta todo para que la pantalla principal sea la lista de medios de pago.

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

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

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
      return MaterialApp(
      title: 'Consumo de API',
      theme: ThemeData(primarySwatch: Colors.blue),
      home: MediosPagoScreen(),
      );
  }
}
```

Resultado esperado

Se mostrará una lista con los medios de pago.

Cada elemento muestra:

- El nombre del medio de pago.
- Su ID y si requiere cargar documentos.
- Un ícono indicando si está activo (✔️) o no (❌)

EJECUTAR LA APLICACIÓN

Ejecutar en Navegador Web

Paso 1: Habilitar Flutter para web

Asegurarse de que Flutter está configurado para desarrollo web. Ejecutar el siguiente comando en el terminal:

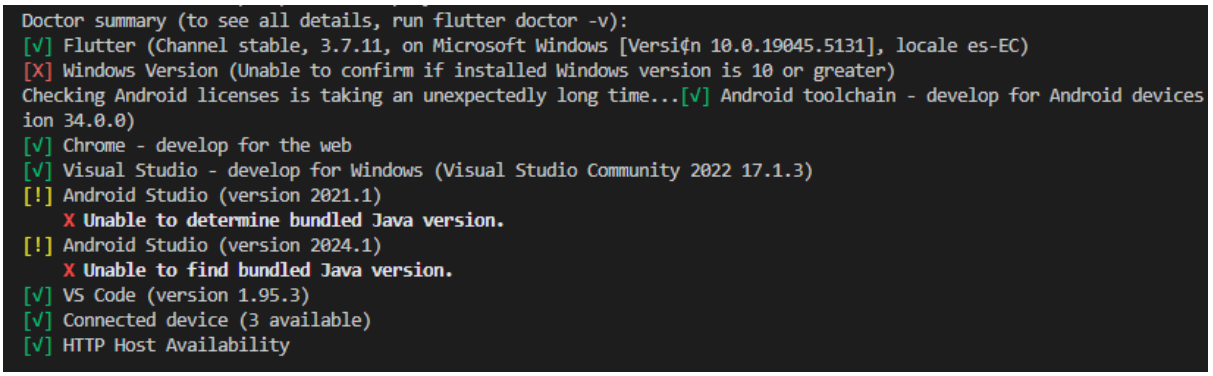
```
flutter config --enable-web
```

Paso 2: Verifica que web está habilitado

Ejecutar:

```
flutter doctor
```

En la salida, debería verse similar a:



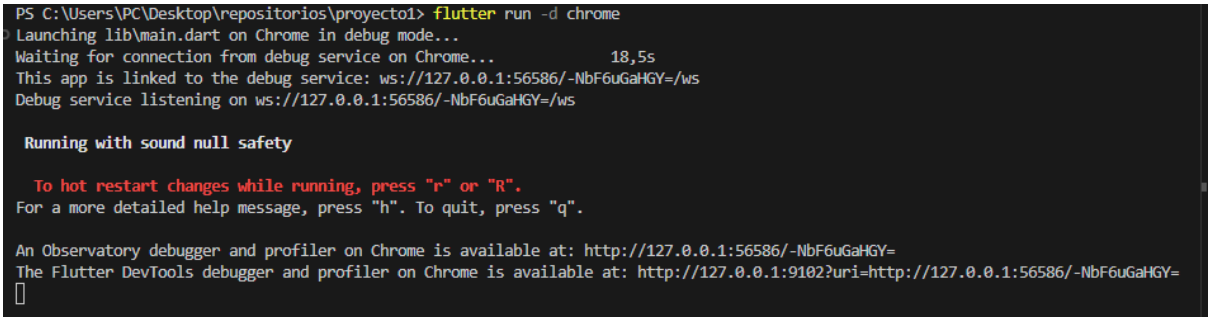
Paso 3: Ejecutar la aplicación en el navegador

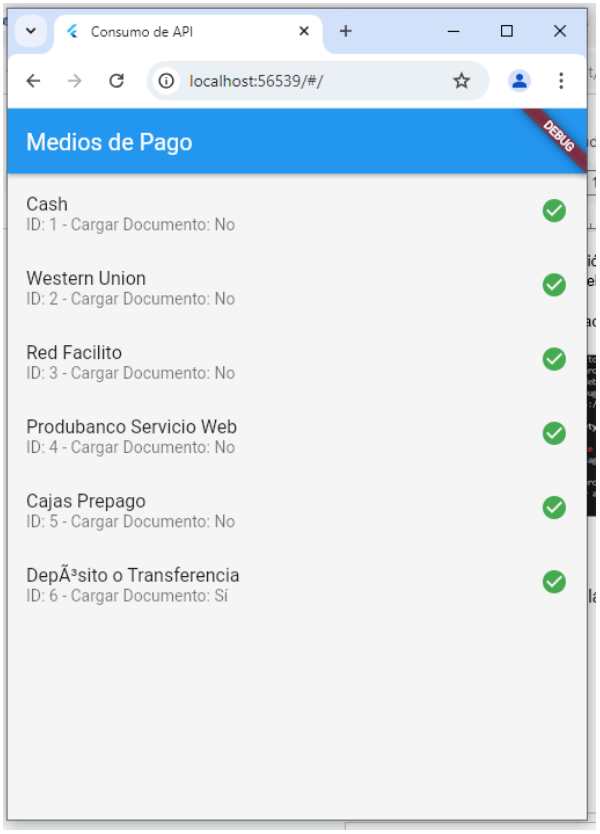
En el directorio del proyecto ejecutar:

```
flutter run -d chrome
```

Esto abrirá tu aplicación en Google Chrome. Si hay otros navegadores instalados y configurados, puede elegirlos desde el terminal.

Al ejecutarse la aplicación el terminal mostrará esta salida si no hay ningún error:





Paso 4: Build para producción

Para crear una versión optimizada para producción:

```
flutter build web
```

Esto generará un directorio `build/web` que se puede subir a un servidor.

Ejecutar en Celular Android o Emulador

Paso 1: Configura tu entorno para Android

Asegurarse de haber instalado y configurado:

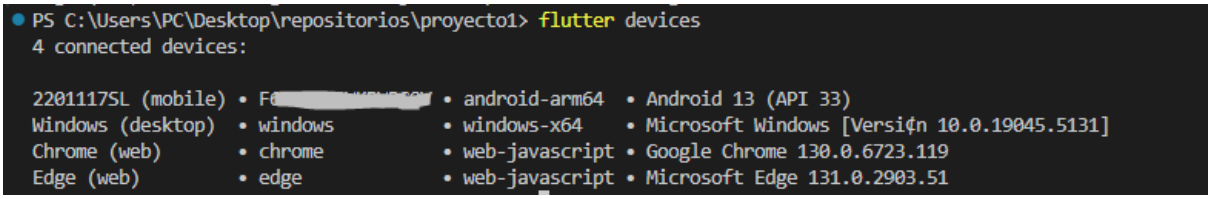
1. **Android Studio**: Configura el SDK de Android y crea un dispositivo virtual (emulador).
2. **Un dispositivo físico Android**: Habilitar la depuración USB en tu teléfono.
    - Ir a **Ajustes > Opciones de desarrollador > Depuración USB** y actívala.

Paso 2: Verifica dispositivos conectados

Conecta tu teléfono Android al ordenador o inicia un emulador y verifica que Flutter lo reconoce:

```
flutter devices
```

Debería ver una lista de dispositivos disponibles, cada uno tiene su ID.



Aquí se muestra el primer dispositivo que es el celular Android y tiene un codigo como por ejemplo FXXXXXXXX.

Paso 3: Ejecuta la aplicación en Android

Si el celular está conectado ejecutar:

```
flutter run -d codigo_de_dispositivo
```

Paso 4: Build para APK
El paso final para poner en producción es la construcción de la apk.

```
flutter build apk
```

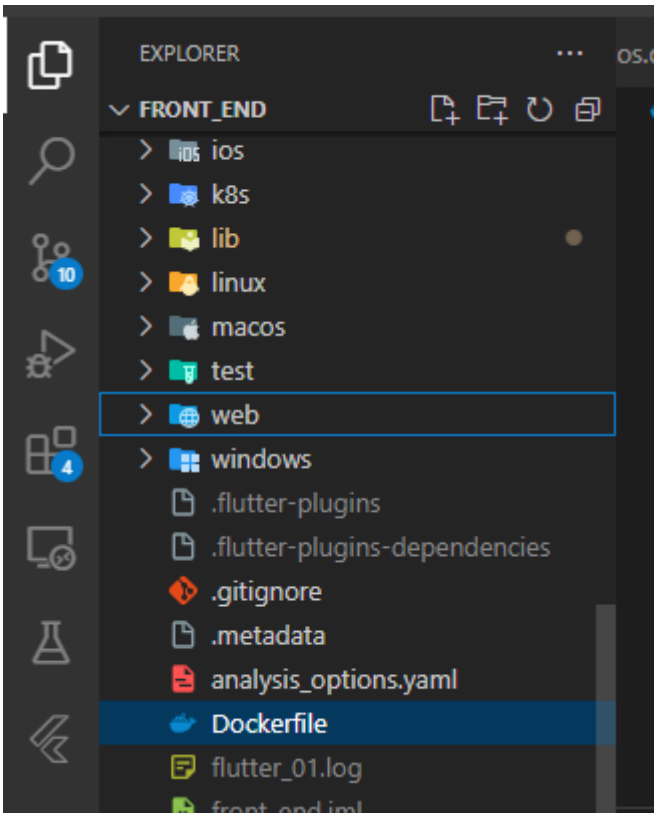
El comando se ejecuta para generar un archivo .apk en el directorio \build\app\outputs\apk\release que puede ser instalado en un dispositivo Android.

CONSTRUIR IMAGEN DOCKER

Para construir la imagen el primero paso es construir el proyecto tanto para web como para móvil con los respectivos comandos

flutter build web
flutter build apk

En la carpeta raíz del proyecto se debe crear un archivo DockerFile



Lo que contiene el archivo es la instalación de un servidor de aplicaciones, en este caso va a ser Apache, luego se procede a copiar los archivos en la carpeta htdocs como en el ejemplo:

```
FROM httpd:2.4
ENV TZ=America/Guayaquil
EXPOSE 80
EXPOSE 8082

COPY build/web /usr/local/apache2/htdocs/
COPY build/app/outputs/apk/release/app-release.apk /usr/local/apache2/htdocs/
RUN mv /usr/local/apache2/htdocs/app-release.apk
/usr/local/apache2/htdocs/saitel_app.apk
```

Los comandos para construir y correr el contenedor son los siguientes:

docker build -t nombre-aplicacion:version .

```
docker build -t app:1.0.1 .
```

Para correr la imagen de manera local y comprobar que esté funcionando correctamente

docker run -p PUERTO --name NOMBRE nombre-app:version

```
docker run -p 80 --name prueba prueba:1.0.1
```

Una vez comprobado el funcionamiento, se procede a guardar un archivo con extensión .tar para subir al servidor

docker save nombre-app:version -o nombre-archivo.tar

```
docker save prueba:1.0.1 -o prueba1.0.1.tar
```

Finalmente cargar la version creada al servidor para que el backend lo coloque en los kubernetes

```
scp -P 65000 prueba1.0.1.tar saitel@192.168.217.29:.
```

ANDROID STUDIO

INTRODUCCIÓN

Android Studio es el entorno de desarrollo integrado (IDE) oficial para el desarrollo de aplicaciones Android. Fue creado por Google y está basado en IntelliJ IDEA, una plataforma para el desarrollo de software. Android Studio ofrece herramientas y características diseñadas específicamente para facilitar el desarrollo, depuración y publicación de aplicaciones Android.

Principales características de Android Studio

1. **Editor de código avanzado:**
    - Soporta los lenguajes **Kotlin**, **Java** y **C++**.
    - Ofrece autocompletado de código, resaltado de errores en tiempo real y sugerencias para optimizar tu código.
2. **Diseñador de interfaces (UI):**
    - Incluye un editor visual que permite crear y personalizar la interfaz gráfica de la aplicación arrastrando y soltando elementos.
    - Proporciona una vista previa en tiempo real para diferentes tamaños de pantalla y dispositivos.
3. **Administrador de dispositivos virtuales (AVD):**
    - Permite emular dispositivos Android para probar aplicaciones sin necesidad de un dispositivo físico.
    - Compatible con múltiples versiones de Android.
4. **Integración con Gradle:**
    - Usa **Gradle** como sistema de construcción, lo que facilita la gestión de dependencias, configuraciones de compilación y publicación de aplicaciones.
5. **Depuración y pruebas:**
    - Ofrece herramientas avanzadas para depuración, como inspección de memoria, análisis de CPU y registro de errores en tiempo real.
    - Soporte para pruebas unitarias y pruebas de interfaz de usuario.
6. **Compatibilidad con herramientas de Google:**
    - Se integra fácilmente con servicios como Firebase, Google Maps y Google Play Console para agregar funciones avanzadas a las aplicaciones.
7. **Actualizaciones constantes:**
    - Google actualiza Android Studio regularmente, incluyendo soporte para nuevas versiones de Android y características innovadoras.

INSTALACIÓN

Los requisitos de un computador para Android Studio son:

- **Sistema operativo**: Microsoft Windows 8/10/11 de 64 bits
- **Procesador**: Procesador Intel Core de segunda generación o posterior, o CPU AMD compatible con un hipervisor de Windows
- **RAM**: 8 GB de RAM o más
- **Espacio de almacenamiento**: 8 GB de espacio disponible en el disco como mínimo

Para Mac, los requisitos son:

- RAM: Mínimo 8 GB, recomendado 16 GB o más
- CPU: Mínimo Chip Apple M1 o Intel Core de segunda generación o posterior, compatible con el framework de hipervisor

- Espacio en el disco: Mínimo 8 GB, recomendado unidad de estado sólido con 16 GB o más

Paso 1: Descargar Android Studio

1. Ir al sitio oficial de Android Studio: https://developer.android.com/studio.
2. Hacer clic en el botón **"Download Android Studio"**.
3. Aceptar los términos y condiciones y descarga el instalador para tu sistema operativo (Windows, macOS o Linux).

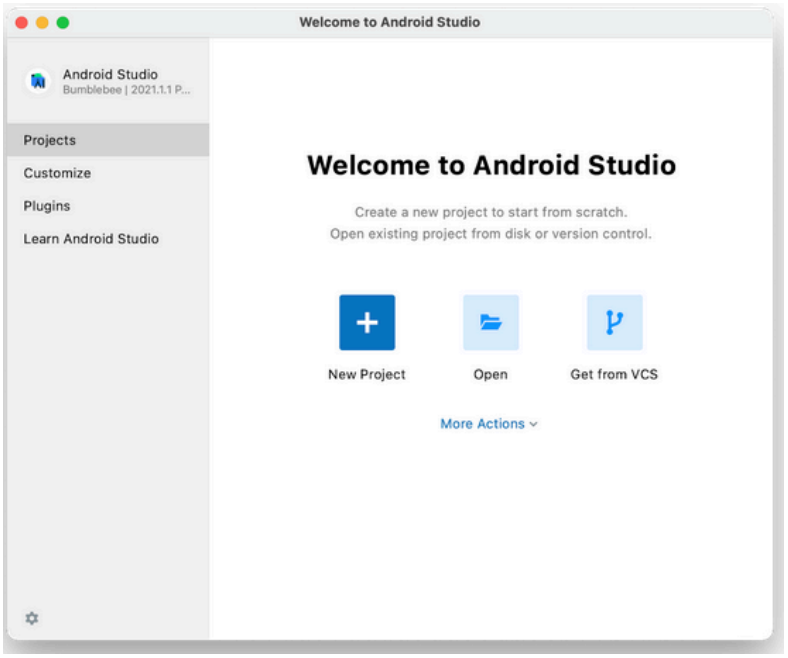
Paso 2: Instalar Android Studio

1. **Windows**:
    - Ejecutar el archivo descargado (`.exe`) y seguir las instrucciones del instalador.
    - Durante la instalación, seleccionar la opción para instalar el **Android SDK**, **Android Virtual Device (AVD)** y las herramientas necesarias.
2. **macOS**:
    - Abrir el archivo `.dmg` descargado y arrastra Android Studio a la carpeta **Applications**.
    - Ábrelo desde Applications y sigue las instrucciones de configuración.
3. **Linux**:
    - Extraer el archivo descargado y ejecutar el script `studio.sh` desde el terminal.
    - Configurar los SDK y complementos necesarios.

CREAR PROYECTO

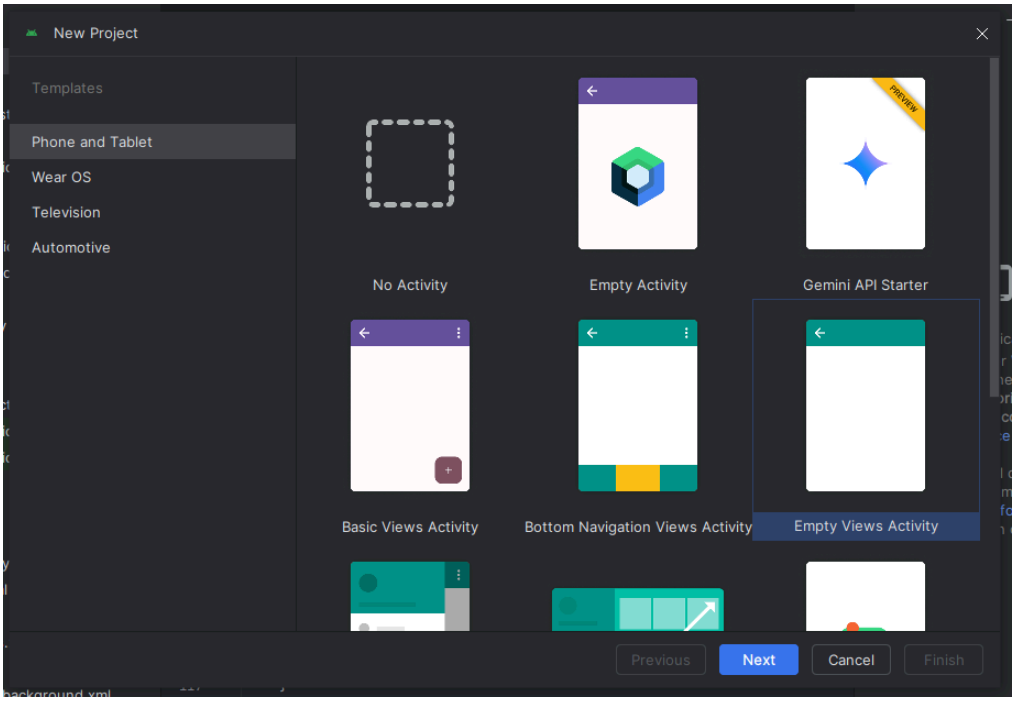
Paso 1: Abrir Android Studio

En la pantalla de bienvenida, seleccionar **"New Project"**.



Paso 2: Elegir plantilla o proyecto vacío

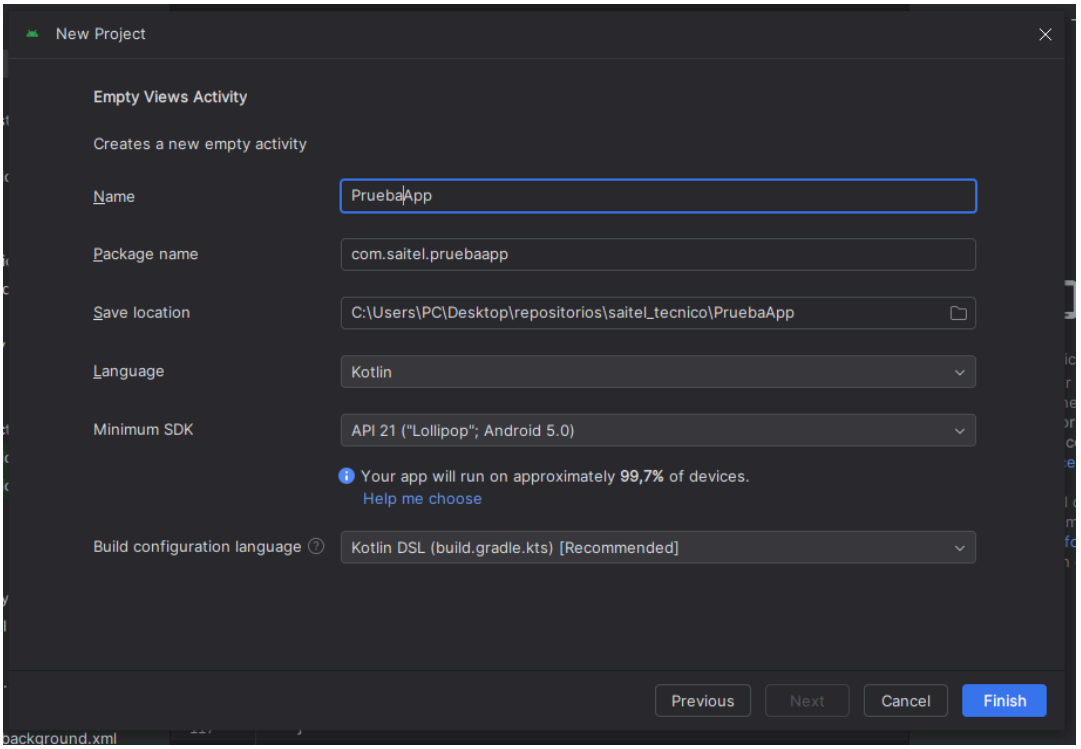
En este caso vamos a seleccionar la opción de **Empty Views Activity** para empezar desde cero el proyecto y hacer click en **Next**



Paso 3: Configurar el proyecto

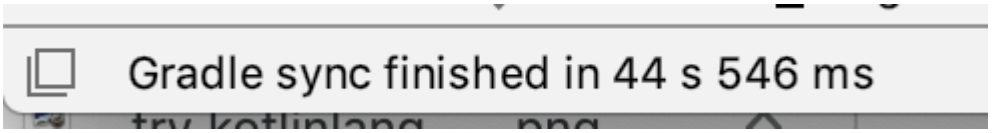
- **Name:** Escribe el nombre de tu aplicación (por ejemplo, `PruebaApp`).
- **Package Name:** Este es el identificador único de tu aplicación (por ejemplo, **`com.ejemplo.pruebaapp`**).

- **Save Location:** Elige la ubicación donde se guardará tu proyecto.
- **Language:** Selecciona Kotlin (recomendado) o Java.
- **Minimum SDK:** Selecciona la versión mínima de Android que soportará tu aplicación (por ejemplo, `API 21: Android 5.0 (Lollipop)`).
- Haz clic en **"Finish".**

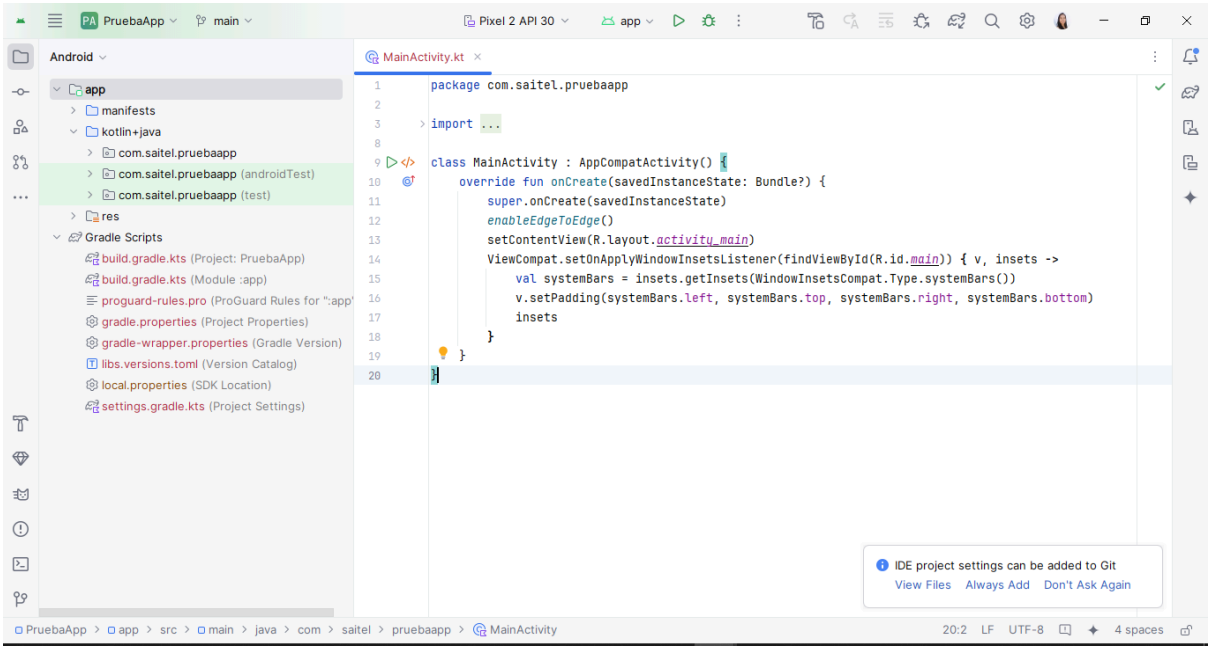


Ese proceso puede tardar un poco. Mientras se configura Android Studio, un mensaje y una barra de progreso indicarán si aún se está configurando tu proyecto. Es posible que se vea de este modo:

Un mensaje similar a éste te informará cuando se cree la configuración del proyecto.



Una vez que el proyecto esté creado se visualizará de esta manera



ESTRUCTURA DEL PROYECTO

Al crear el proyecto se genera una estructura principal del proyecto como la siguiente:

```
PruebaApp/
├── .idea/
├── app/
│   ├── build/
│   ├── src/
│   │   ├── main/
│   │   │   ├── java/
│   │   │   ├── res/
│   │   │   └── AndroidManifest.xml
│   │   ├── test/
│   │   └── androidTest/
│   └── build.gradle
├── gradle/
├── build.gradle
└── settings.gradle
```

1. Carpeta raíz del proyecto

- **`PruebaApp/`**: Es la carpeta principal que contiene todos los archivos y configuraciones del proyecto.
- **`build.gradle` (a nivel raíz):** Configuración general del proyecto, como la versión de Gradle, dependencias comunes y repositorios.
- **`settings.gradle`:** Define los módulos incluidos en el proyecto (por ejemplo, `app`).

2. Módulo principal: `app/`

El módulo `app/` contiene todo lo relacionado con tu aplicación.

a) `src/`

- **`src/main/`**: Donde resides el código fuente y los recursos principales de tu aplicación.
    - **`java/`:**
      Contiene el código fuente de tu aplicación escrito en Kotlin o Java.
      Por ejemplo:

      ```
      app/src/main/java/com/ejemplo/pruebaapp/MainActivity.kt
      ```

    - **`res/`:**
      Contiene los recursos de tu aplicación, organizados en subcarpetas:
        - **`layout/`:** Archivos XML para definir la interfaz de usuario (UI).
          Ejemplo: `activity_main.xml`.
        - **`drawable/`:** Imágenes y formas gráficas utilizadas en la app.
        - **`values/`:** Archivos XML con valores reutilizables, como:
            - **`strings.xml`:** Textos usados en la app (para soportar múltiples idiomas).
            - **`colors.xml`:** Definición de colores.
            - **`styles.xml`:** Estilos y temas de la app.
        - **`mipmap/`:** Iconos de la aplicación en diferentes resoluciones.
    - **`AndroidManifest.xml`:**
      Archivo central donde defines:
        - Información de la aplicación (nombre, icono, tema, permisos).
        - Declaración de actividades, servicios y receptores.
- **`src/test/`:**
  Archivos para pruebas unitarias (lógica del negocio).
- **`src/androidTest/`:**
  Archivos para pruebas de interfaz y funcionalidad en Android.

b) `build/`

- Contiene los archivos generados automáticamente al compilar el proyecto.

c) `build.gradle` (a nivel de `app/`):

- Configuración específica del módulo, como:
  - Dependencias (librerías externas).
  - Configuración del SDK (versión mínima y máxima).
  - Configuración de firma para publicar tu aplicación.

3. Otras carpetas importantes

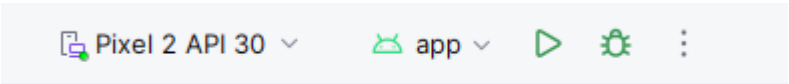
- **`.idea/`:**
  Archivos de configuración específicos de Android Studio (no necesitas editarlos manualmente).
- **`gradle/`:**
  Archivos de configuración de Gradle que administran el sistema de construcción del proyecto.

Con esta estructura general el flujo de trabajo sería:

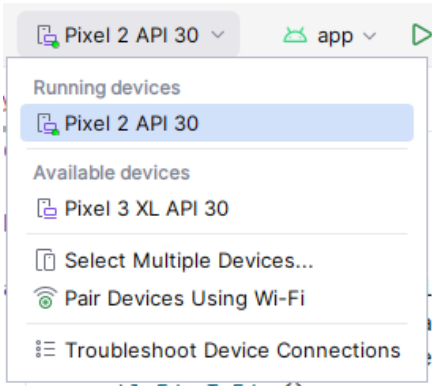
1. Escribir el código fuente en la carpeta **`java/`** o **`kotlin/`**.
2. Diseñar las interfaces en **`res/layout/`**.
3. Declarar actividades, permisos y configuraciones en **`AndroidManifest.xml`**.
4. Gradle compila el código y genera la APK o AAB para distribuir.

EJECUCIÓN DE APLICACIÓN

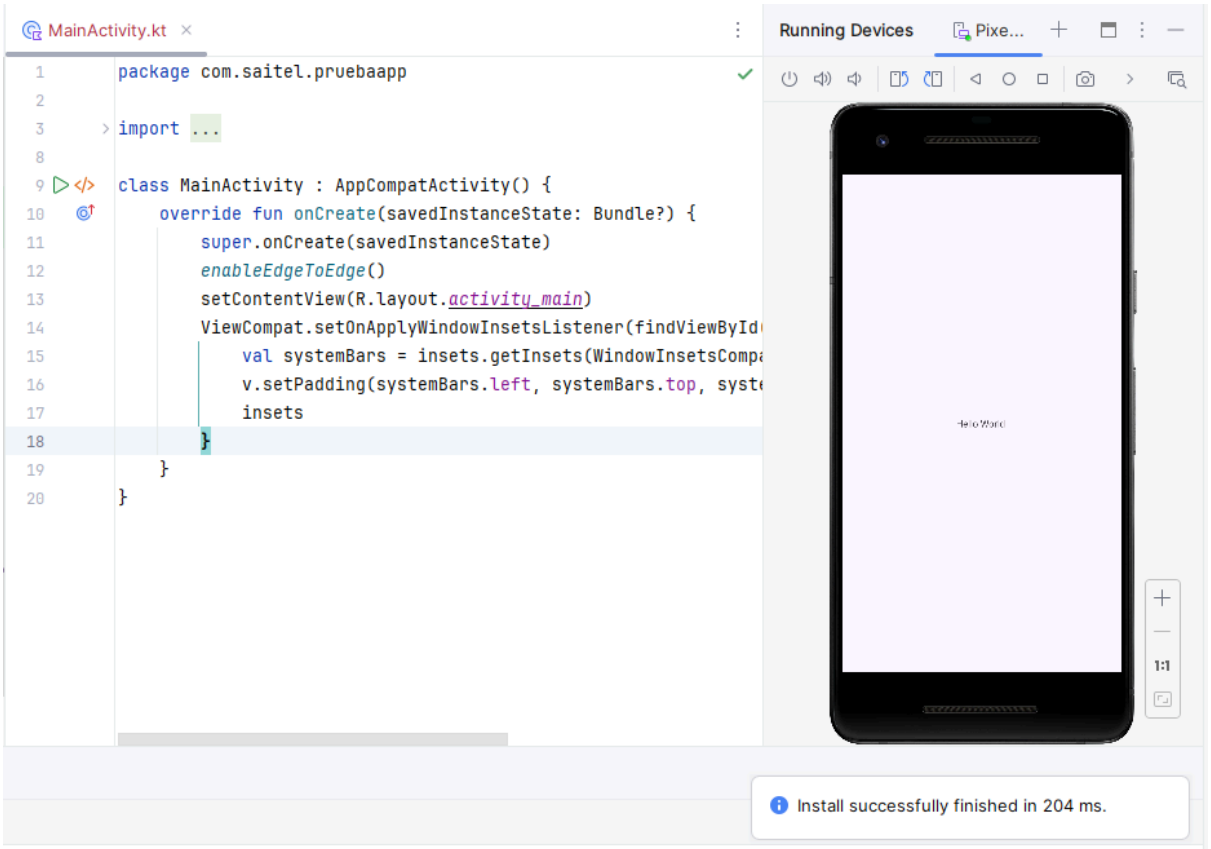
En la parte superior tenemos botones para correr la aplicación, uno de ellos es el ícono de **Run app** ▷



También tenemos una lista de dispositivos, que muestran los que el emulador tiene disponible y en caso de conectar un dispositivo físico Android de igual manera aparecerá en la lista para elegirlo al momento de correr la app.

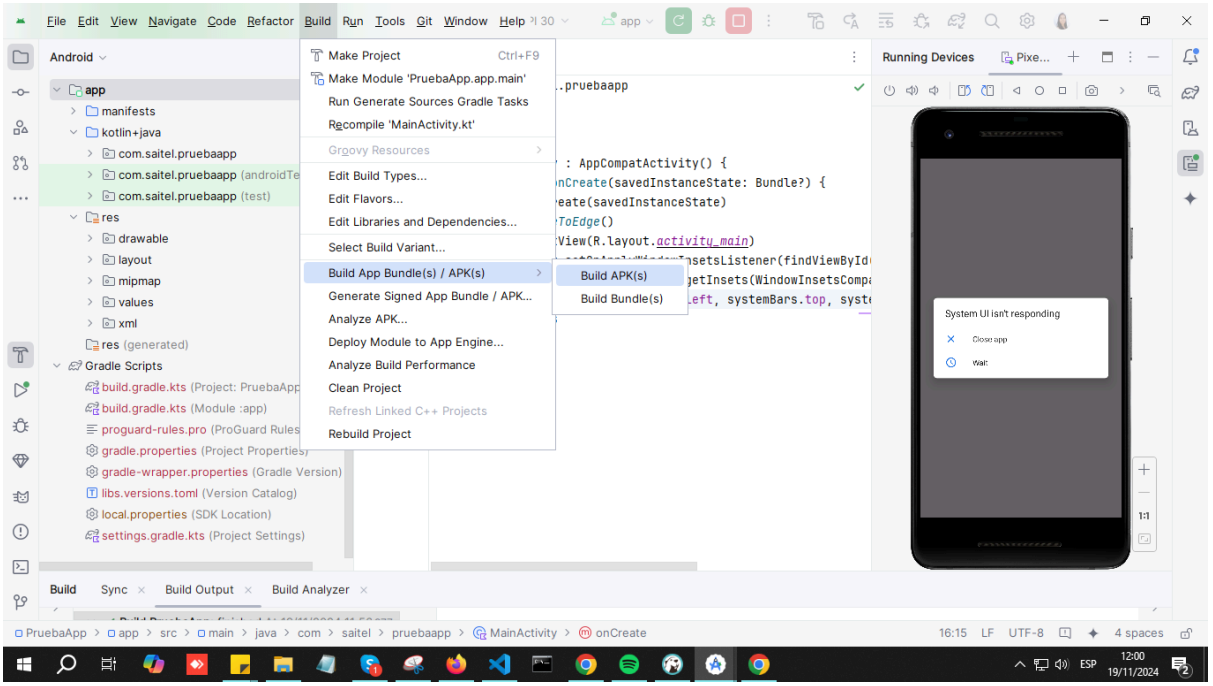


Ejemplo de la aplicación corriendo en el emulador:

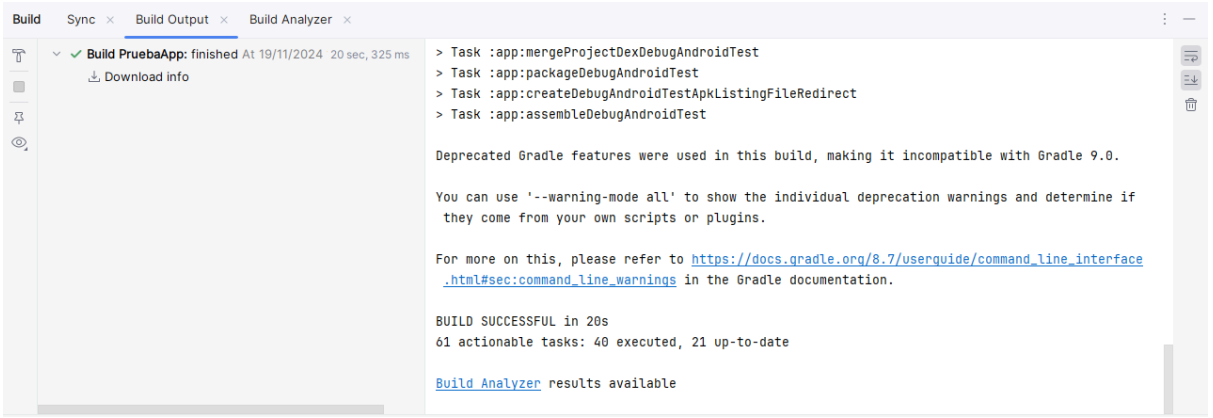


CONSTRUIR APK

En la barra superior nos dirigimos al item de Build y generamos el APK



En caso no haber errores se mostrará en la consola un mensaje de éxito



Creando la APK en el directorio `PruebaApp\app\build\outputs\apk\debug`

EJEMPLO DE APLICACIÓN

Estructura del proyecto:

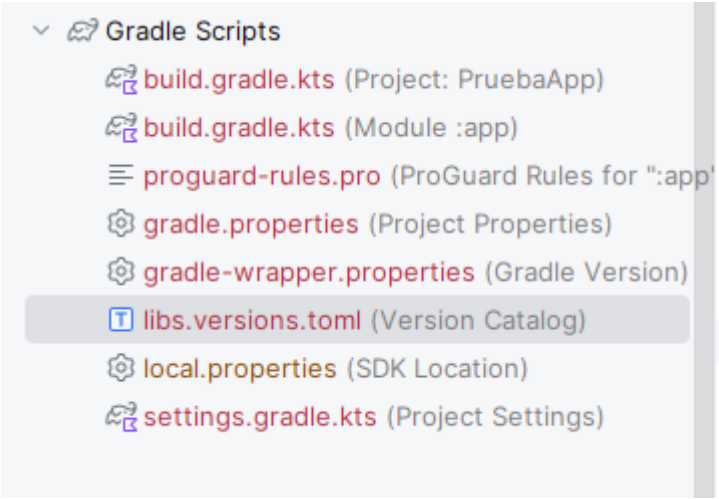
```
MyApiApp/
├── app/
│   ├── src/
│   │   ├── main/
│   │   │   ├── java/
│   │   │   │   └── com.example.myapiapp/
│   │   │   │   ├── MainActivity.kt
│   │   │   │   ├── ApiSingleton.kt
│   │   │   │   ├── ApiAdapter.kt
│   │   │   │   └── models/
│   │   │   │         └── ApiResponse.kt
│   │   │   ├── res/
│   │   │   │   ├── layout/
│   │   │   │   │   ├── activity_main.xml
│   │   │   │   │   └── item_post.xml
│   │   │   │   ├── values/
│   │   │   │   │   ├── colors.xml
│   │   │   │   │   ├── strings.xml
│   │   │   │   │   └── styles.xml
│   │   │   └── AndroidManifest.xml
│   └── build.gradle
├── build.gradle
└── settings.gradle
```

Configuración de las dependencias:

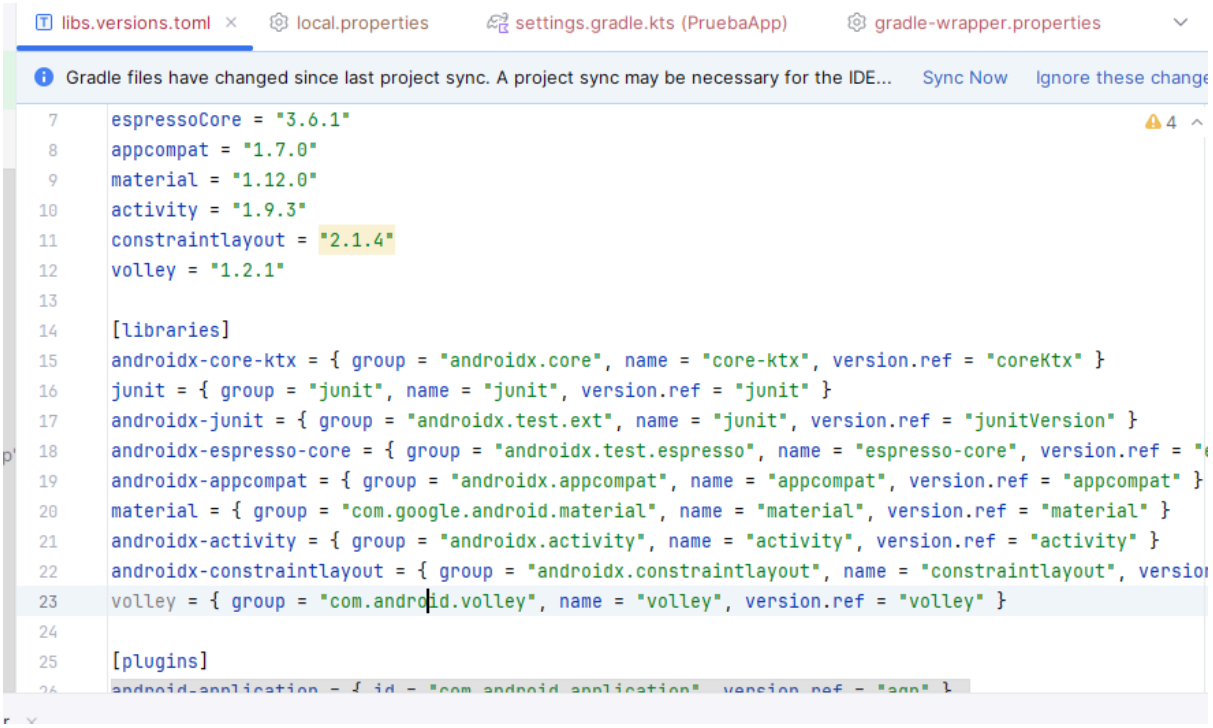
Agregar la dependencia de Volley en el archivo `build.gradle`

```
38  dependencies {
39
40      implementation(libs.androidx.core.ktx)
41      implementation(libs.androidx.appcompat)
42      implementation(libs.material)
43      implementation(libs.androidx.activity)
44      implementation(libs.androidx.constraintlayout)
45      testImplementation(libs.junit)
46      implementation(libs.volley)
47      androidTestImplementation(libs.androidx.junit)
48      androidTestImplementation(libs.androidx.espresso.core)
49  }
```

Como estamos usando Version Catalogs para gestionar las dependencias, asegurarse de que la la librería de encuentre en **libs.version.toml**



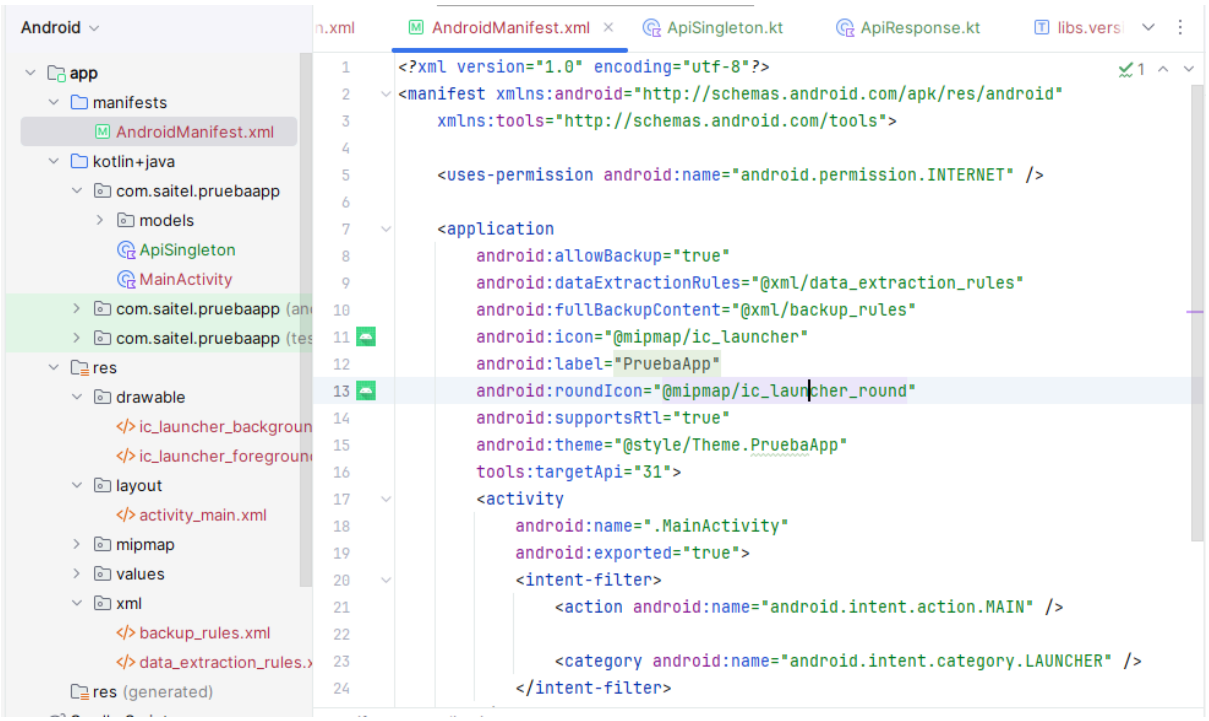
Si no existe el registro de la librería, se debe agregar las líneas correspondientes a la librería volley:



Configuración de AndroidManifest.xml

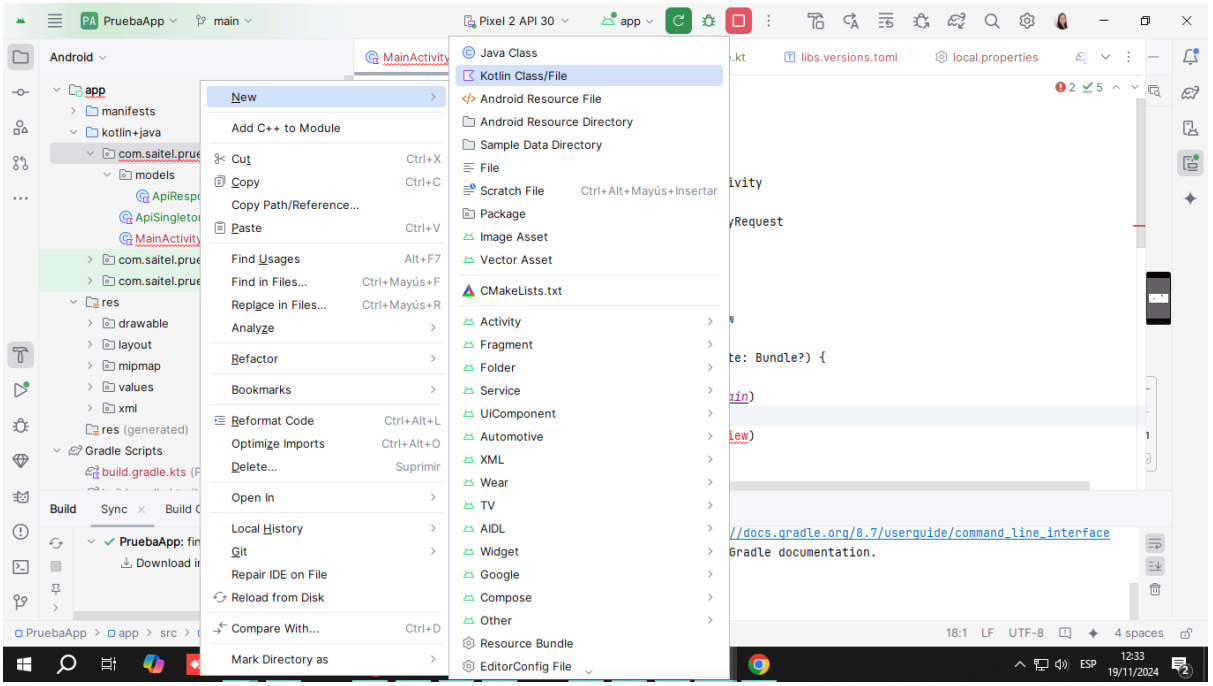
Como vamos a consumir una API necesitamos habilitar la conexión a internet a la aplicación por lo tanto agregamos la línea

```
<uses-permission android:name="android.permission.INTERNET" />
```



Creación de los archivos necesarios de acuerdo a la estructura que se ha presentado:

Dar click en la carpeta raíz e ir creando los archivos como Kotlin/ Class File



a) MainActivity.kt

El archivo principal de la actividad donde consumimos la API y mostramos los datos:

```
package com.saitel.pruebaapp

import android.os.Bundle
import android.widget.TextView
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity
import androidx.recyclerview.widget.LinearLayoutManager
import androidx.recyclerview.widget.RecyclerView
import com.android.volley.Request
import com.android.volley.toolbox.JsonArrayRequest
import com.saitel.pruebaapp.models.ApiResponse
```

```
class MainActivity : AppCompatActivity() {

    private lateinit var recyclerView: RecyclerView
    private val posts = mutableListOf<ApiResponse>()

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

        recyclerView = findViewById(R.id.recyclerView)
        recyclerView.layoutManager = LinearLayoutManager(this)

        fetchPosts()
    }

    private fun fetchPosts() {
        val url = "https://jsonplaceholder.typicode.com/posts"

        val jsonArrayRequest = JsonArrayRequest(
            Request.Method.GET, url, null,
            { response ->
                for (i in 0 until response.length()) {
                    val post = response.getJSONObject(i)
                    posts.add(
                        ApiResponse(
                            id = post.getInt("id"),
                            title = post.getString("title"),
                            body = post.getString("body")
                        )
                    )
                }
                recyclerView.adapter = ApiAdapter(posts)
            },
            { error ->
                Toast.makeText(this, "Error: ${error.message}",
Toast.LENGTH_SHORT).show()
            }
        )
```

```
ApiSingleton.getInstance(this).addToRequestQueue(jsonArrayRequest
)
    }
}
```

b) ApiSingleton.kt

Una clase singleton para manejar la cola de solicitudes de Volley.

```
package com.saitel.pruebaapp

import android.content.Context
import com.android.volley.Request
import com.android.volley.RequestQueue
import com.android.volley.toolbox.Volley

class ApiSingleton private constructor(context: Context) {
    companion object {
        @Volatile
        private var INSTANCE: ApiSingleton? = null

        fun getInstance(context: Context) =
            INSTANCE ?: synchronized(this) {
                INSTANCE ?: ApiSingleton(context).also { INSTANCE
= it }
            }
    }

    private val requestQueue: RequestQueue by lazy {
        Volley.newRequestQueue(context.applicationContext)
    }

    fun <T> addToRequestQueue(req: Request<T>) {
        requestQueue.add(req)
    }
}
```

c) ApiResponse.kt

Para estructurar la respuesta de la API en un modelo de datos.

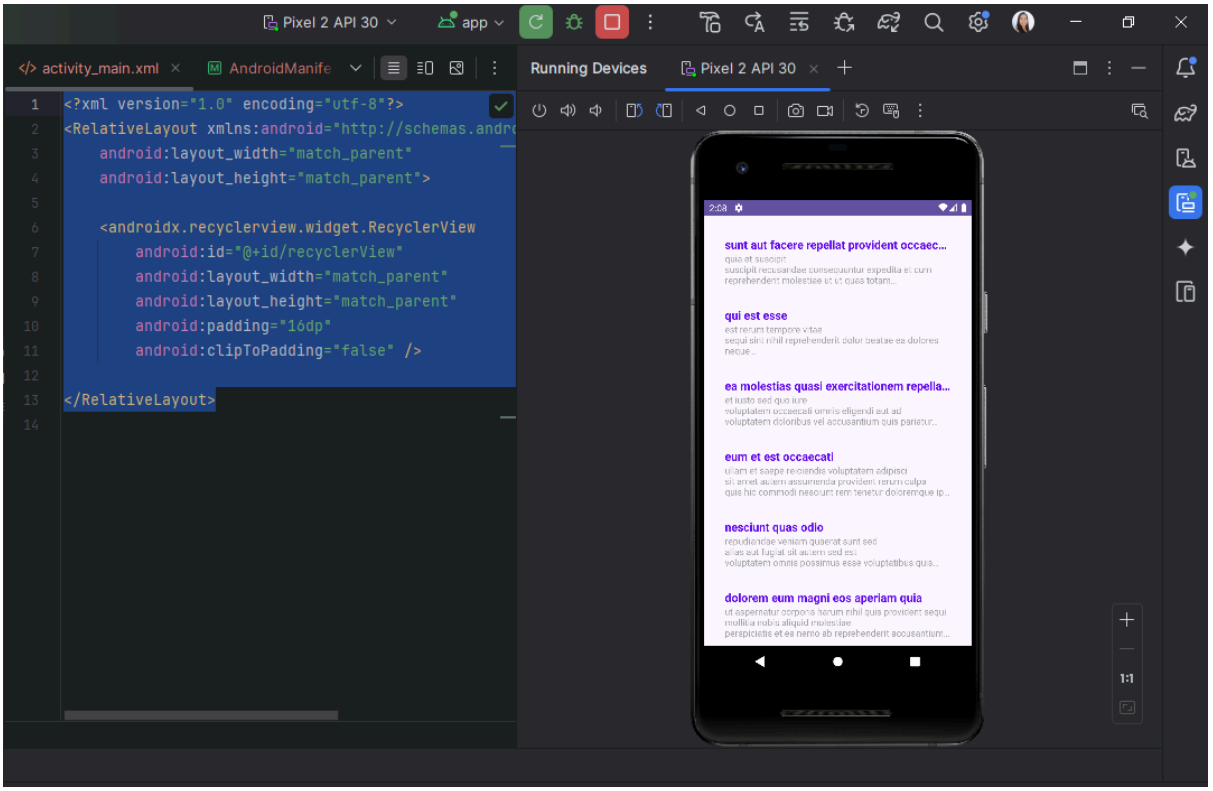
```
package com.saitel.pruebaapp.models

data class ApiResponse(
    val id: Int,
    val title: String,
    val body: String
)
```

d) activity_main.xml

Interfaz gráfica básica que muestra los datos.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/recyclerView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:padding="16dp"
        android:clipToPadding="false" />

</RelativeLayout>
```

Resultado al correr la aplicación en el emulador:


Pixel 2 API 30
app
activity_main.xml
AndroidManife
Running Devices
Pixel 2 API 30
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.andr
android:layout_width="match_parent"
android:layout_height="match_parent">
<androidx.recyclerview.widget.RecyclerView
android:id="@+id/recyclerView"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:padding="16dp"
android:clipToPadding="false" />
</RelativeLayout>
sunt aut facere repellat provident occaec...
qui est esse
ea molestias quasi exercitationem repella...
eum et est occaecati
nesciunt quas odio
dolorem eum magni eos aperiam quia
1:1