



Maria Ana Sanz

Jonathan Del Arco
2º DAM

Roomfy App

16 de Mayo del 2023

[Enlace al repositorio](#)

ÍNDICE

1. Objetivos.....	4
1.1 Descripción.....	4
1.2 Funcionalidades.....	4
2. Recursos de hardware y software.....	5
2.1 Hardware.....	5
2.2 Software.....	5
3. Enumeración y desarrollo de las fases.....	6
3.1 Análisis.....	6
3.1.1 Esquema general.....	6
3.1.2 Diagrama de secuencia - Registro.....	7
3.1.3 Diagrama de secuencia - Login.....	8
3.1.4 Diagrama de secuencia - Mensajería.....	9
3.2 Diseño.....	10
3.2.1 Diseño pantalla de Login.....	10
3.2.2 Diseño pantalla Sign Up.....	11
3.2.3 Diseño pantalla de Chat.....	12
4.3 Desarrollo.....	13
4.3.1 Instalación de herramientas necesarias.....	13
a. Instalación de Visual Studio Code.....	13
b. Instalación de Python.....	14
c. Instalación de Dart y Flutter.....	15
d. Instalación de un emulador Android.....	19
e. Acceso al Servidor Amazon AWS.....	25
Instalación de MongoDB en una distribución Ubuntu.....	25
4.3.2 Creando una comunicación entre Flutter y Flask Socket IO.....	27
a. La parte del servidor (localhost).....	27
b. La parte del cliente.....	28
4.3.3 Creando una conexión a MongoDB desde Flask.....	30
a. Creación de la BBDD.....	30
b. Importar instancia a Flask y creación de colecciones.....	31
4.3.4 Realizando el registro de un usuario.....	31
a. register(String json).....	32
b. isRegister().....	33
4.3.5 Realizando el login de usuario.....	35
4.3.6 Enviando y recibiendo mensajes.....	36
a. Providers.....	36
4.3.7 Persistencia de chat.....	40
a. Guardando mensajes en MongoDB.....	40
b. Recuperando mensajes desde MongoDB.....	40
4.4 Deploy.....	42
4.4.1 Cambios en la parte del servidor.....	42
4.4.2 Cambios en la parte del cliente.....	43
4.4.3 Build y APK.....	43
4. Conclusiones.....	44
5. Bibliografía.....	45

Objetivos

Descripción

Roomfy es una aplicación de chat en tiempo real que permite una comunicación segura y rápida entre dos dispositivos móviles a través de salas. Esta aplicación es la respuesta a que es posible una comunicación entre personas sin ofrecer datos sensibles como las grandes compañías cuando nos piden un número de teléfono obligatorio para poder hablar.

Funcionalidades

Las principales funcionalidades son:

- Comunicación entre dos dispositivos móviles

Gracias a los sockets la comunicación entre dispositivos móviles es en tiempo real y rápida.

- Sala de chat grupal

Sala de chat global que agrupa a los usuarios conectados y permite una comunicación entre ellos.

- Seguridad

En un futuro, mensajes encriptados en MongoDB para una mejor seguridad

- UI intuitiva

Se ha diseñado una interfaz aplicando conceptos de usabilidad UX para una mejor experiencia

El objetivo final del proyecto es haber creado una aplicación de comunicación básica que a futuro se pueda escalar añadiendo más funcionalidades sobre todo en el ámbito de la seguridad. Como por ejemplo salas encriptadas. Personalización del chat como su fuente, color de la burbuja del mensaje, etc. En resumen, ofrecer a los usuarios una alternativa a la comunicación de manera gratuita y personalizable pero que en primer lugar este la seguridad y la protección de datos personales.

Recursos de hardware y software

Hardware

Para la realización de este proyecto se ha requerido del siguiente Hardware:

PC:

- Windows 11
- SSD 500GB
- 16GB RAM
- GTX1060 6GB

Móvil Android físico para comprobar el funcionamiento de la App:

- Android 12
- Procesador Snapdragon 680
- Almacenamiento interno 256GB
- 6GB RAM
- LCD 6.5 pulgadas

Software

En cuanto al Software se ha dividido en varios apartados:

- Documentación:
 - Google Docs
- Desarrollo del Software:
 - Frontend:
 - Dart y Flutter como lenguaje y Framework
 - Figma (Diseño de prototipos)
 - Backend:
 - Servidor AWS + MongoDB
 - Flask Socket IO
 - Python 3.11
 - Control de versiones:
 - GitHub
 - Pruebas:
 - Dos emuladores Android

Enumeración y desarrollo de las fases

Análisis

En este proceso se pretende hacer algo más tangible la idea que se tiene en mente y lo que hará la aplicación. Se dividirá el desarrollo en partes y se generarán los diagramas oportunos.

Se ha escogido Flutter porque es un Framework nativo de Android creado por Google, lo que me da bastantes ventajas como el ahorro de tiempo en crear interfaces gráficas ya que disponen de un diseño propio llamado Material 3. Creado específicamente para la usabilidad del usuario. También depende mucho de creación de carpetas lo que da un plus para mantener los proyectos organizados, exportar Widgets, modificarlos, re-utilizarlos, etc.

MongoDB debido a que si se escalara esta aplicación a un futuro y la cantidad de mensajes es elevada, se necesitaría de una buena velocidad y esto solo lo puede otorgar las bases de datos no relacionales.

Flask Socket IO porque es una minimal API que me permite crear una conexión full-duplex a través de sockets, esencial para realizar una aplicación de comunicación en tiempo real.

Esquema general

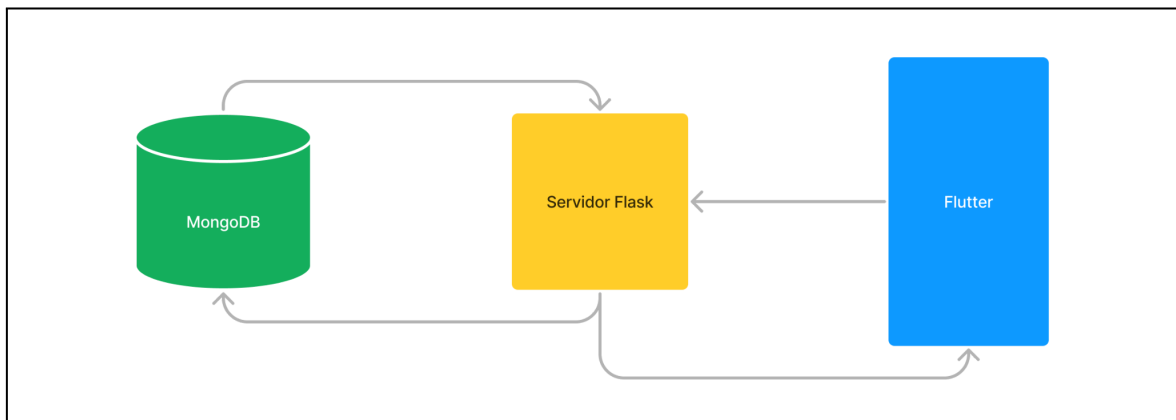


Ilustración 1: Esquema general de cómo está distribuido el entorno

Diagrama de secuencia - Registro

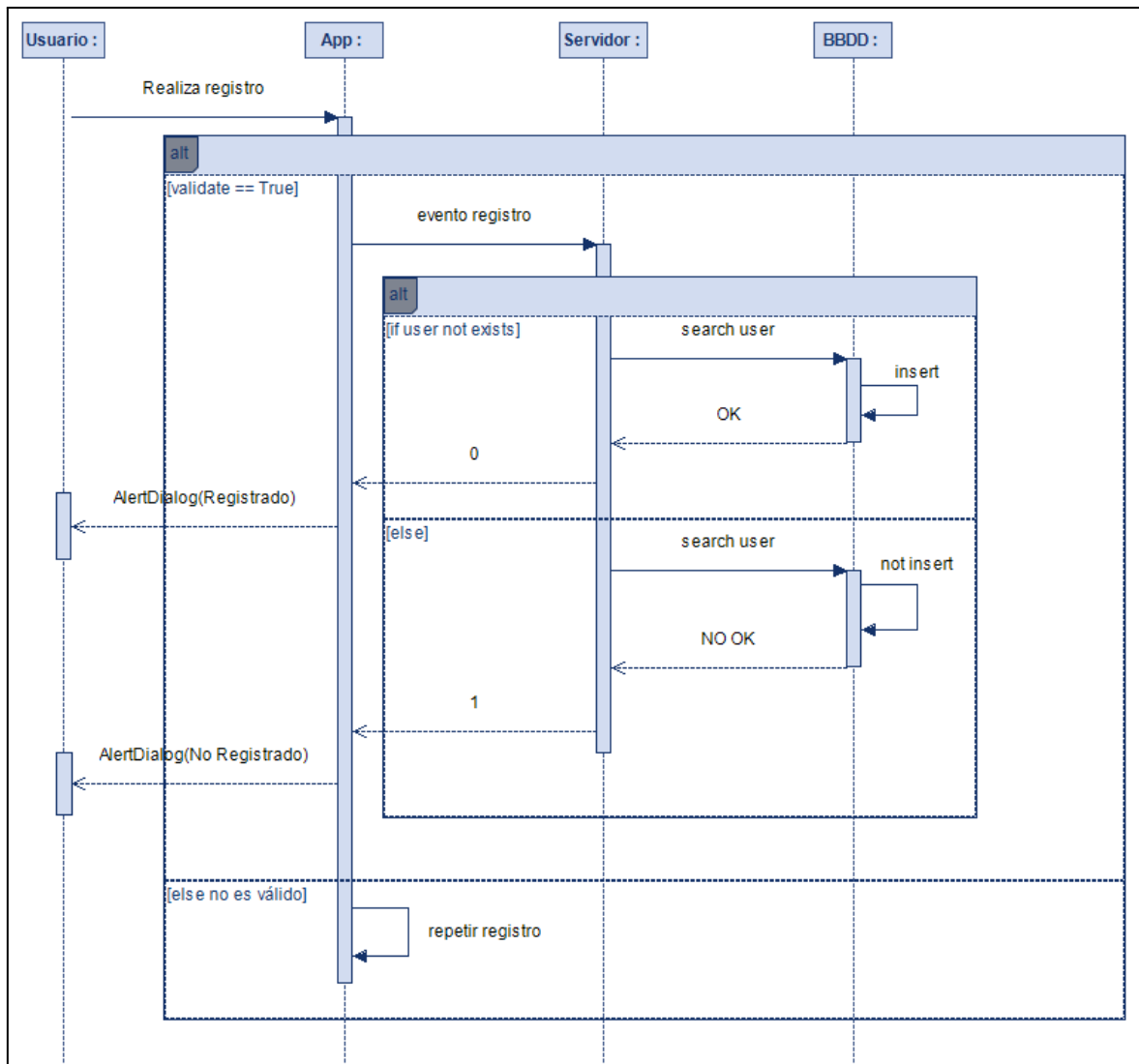


Ilustración 2: Diagrama de secuencia que recorre la pantalla de registro

Diagrama de secuencia - Login

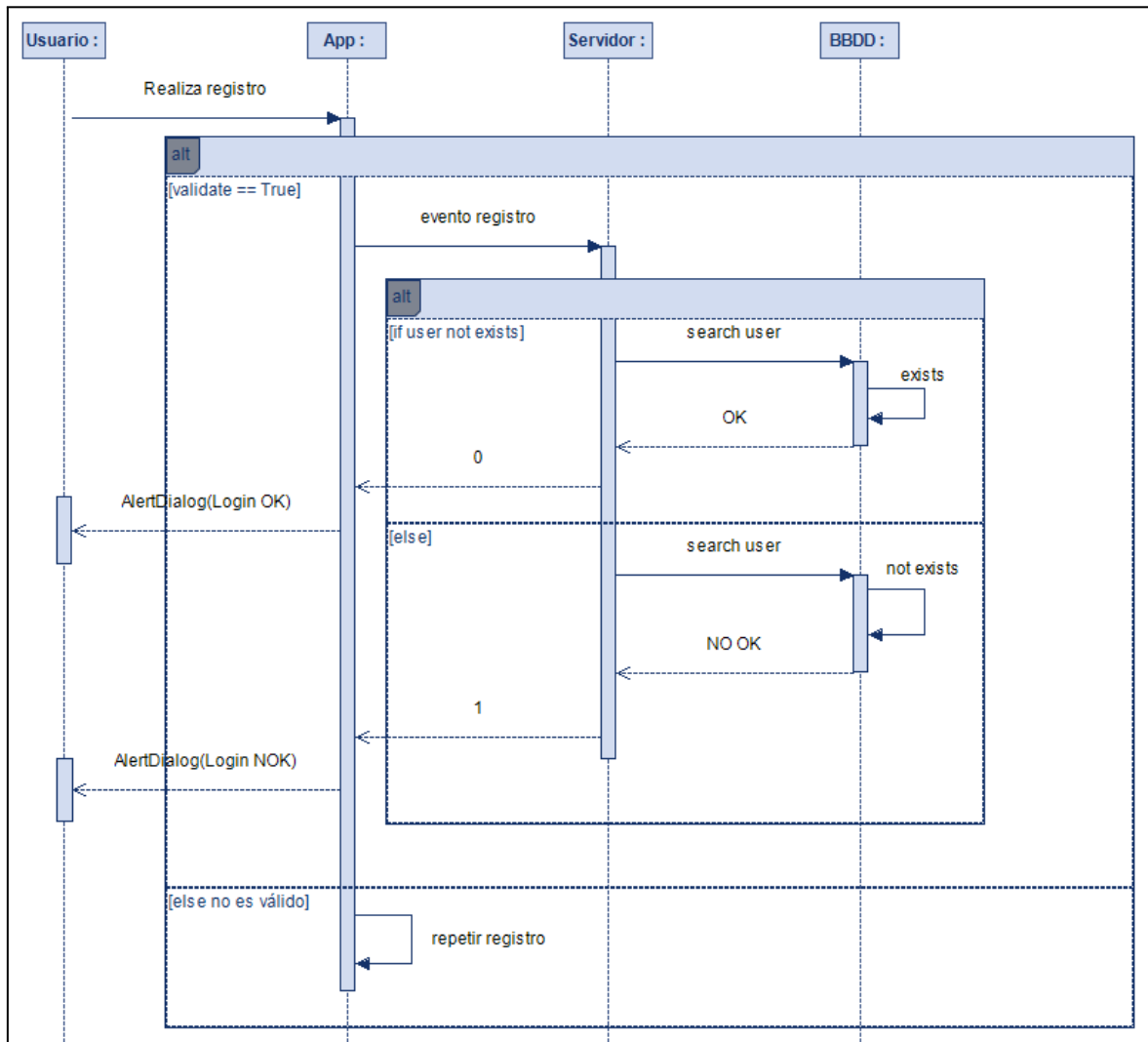


Ilustración 3: Diagrama de secuencia que recorre la pantalla de login

Diagrama de secuencia - Mensajería

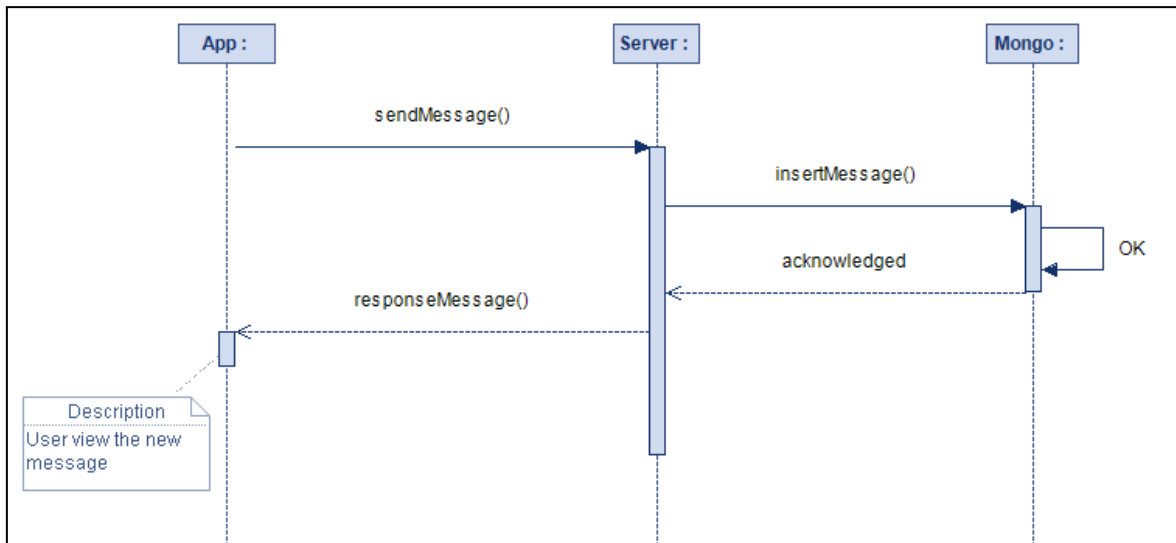


Ilustración 4: Diagrama de secuencia que recorre un mensaje

Diseño

Aquí la idea es diseñar el prototipo de la interfaz que se va a programar en Flutter teniendo las funcionalidades en mente. El Mockup final se hará en Figma. Con una idea clara es más rápido su desarrollo.

Diseño pantalla de Login

El mockup muestra una interfaz de usuario para una pantalla de inicio de sesión. En la parte superior hay un icono de un sobre gris con una 'X' negra. Debajo, el título 'Inicia sesión' está en negrita. Se encuentran dos campos de entrada de texto con el placeholder 'Nombre' y 'Usuario'. Un botón azul con el texto 'Iniciar sesión' está centrado. En la parte inferior, el texto 'Crear nueva cuenta' sirve como enlace. El todo está dentro de un recuadro con esquinas redondeadas.

Inicia sesión

Nombre


Usuario

Iniciar sesión

Crear nueva cuenta

Ilustración 5: Diseño desarrollado en Figma

Diseño pantalla Sign Up



The image shows a mobile app screen for registration. At the top left is a back arrow icon. Next to it is the title 'Registro'. Below the title is the text 'Los usuarios te conocerán como:'. Underneath that is a text input field containing the placeholder 'nombre@usuario'. Below the input field are two more input fields: one labeled 'Nombre' and one labeled 'Usuario'. At the bottom of the form is a blue button with the text 'Registrarme'.

Ilustración 6: Diseño desarrollado en Figma

Diseño pantalla de Chat

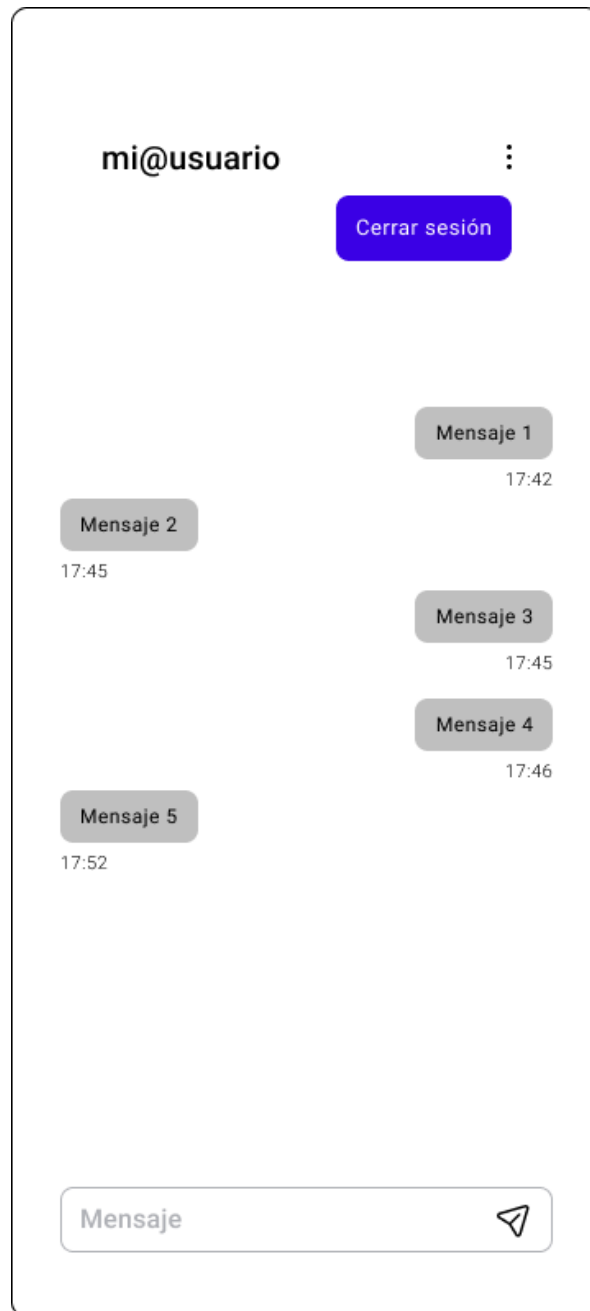


Ilustración 7: Diseño desarrollado en Figma

Desarrollo

En esta parte se comenzará la programación de la aplicación. Por cada parte se irán subiendo capturas del proceso y explicaciones.

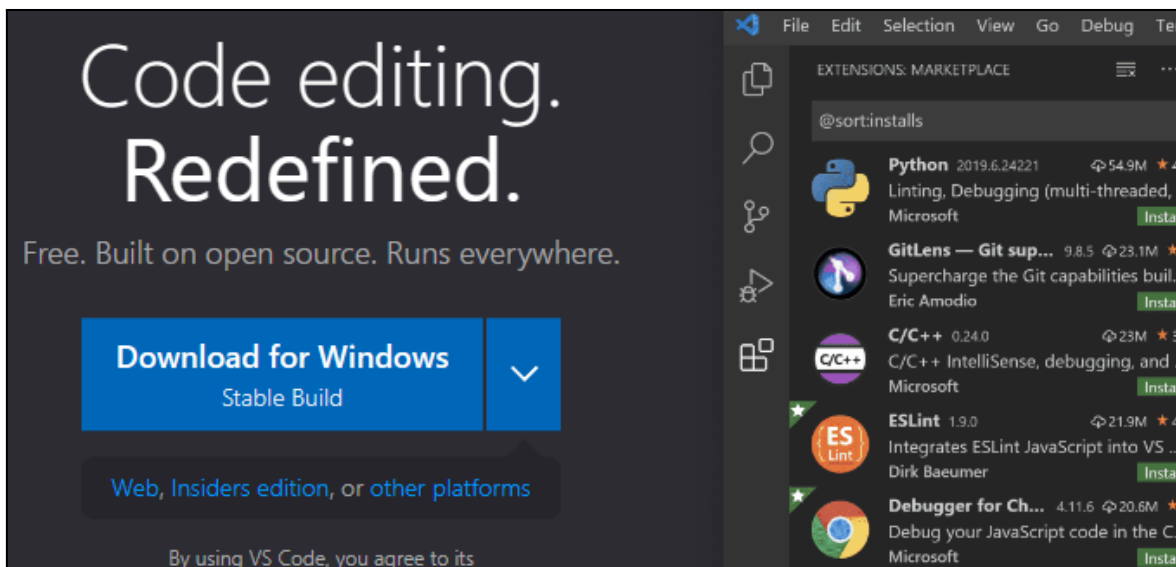
Instalación de herramientas necesarias

En esta parte se explicará la puesta en marcha de todas las herramientas necesarias para desarrollar la aplicación.

Instalación de Visual Studio Code

Lo primero que se ha instalado es el IDE, la herramienta base para poder programar. He decido utilizar VS Code porque es rápido y me ofrece muchas ventajas, como las extensiones que me ahorrarán trabajo. Ahora sólo explicaré su instalación pero más adelante explicaré las extensiones que he ido instalando.

Para ello nos dirigimos a la página web de <https://code.visualstudio.com/> y descargamos la build estable de Windows aquí:



Seguimos las instrucciones de instalación y se abrirá automáticamente el IDE. Se instalará el paquete de idioma español y ya estará listo para su uso.

Instalación de Python

Para instalar Python simplemente me he dirigido a la página web oficial de Python y en la sección 'Downloads' he descargado el instalador.

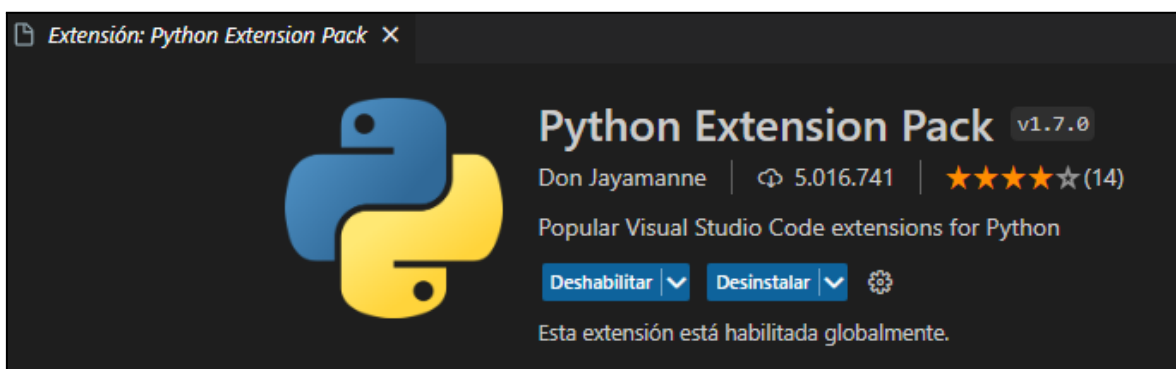


Después he seguido el proceso de instalación tradicional. El instalador en Windows ya se encarga de forma automática de establecer las variables de entorno necesarias para poder trabajar. No así en Linux, que habrá que seguir unos pasos adicionales si queremos instalar la última versión. Por defecto en Linux Python ya viene instalado a la versión 3.9.

Aquí instalaré las primeras extensiones para Python en VS Code. Instalé el Python Extension Pack, que como indica es un paquete de extensiones esenciales para desarrollar en VS Code con Python. Se puede encontrar disponible para descargar aquí:

[Python Extension Pack - Visual Studio Marketplace](#)

Para instalarlo bastará con buscar por el nombre de la extensión e instalar:



Con todo esto tendremos nuestra parte de Python.

Instalación de Dart y Flutter

La instalación de Dart y Flutter es un poco más complicada que las instalaciones anteriores. Ya que como es un lenguaje y un framework todavía joven y en desarrollo puede surgir algún imprevisto. Además de que nos pide software extra dependiendo para qué plataformas vayamos a desarrollar. Para instalar Dart y Flutter tenemos dos caminos. Uno el difícil y otro el fácil.

La instalación difícil es instalar por separado Dart y Flutter. Para mí personalmente esto puede ser inestable por tema de versiones, porque a la hora de realizar el update tendremos que ir primero a por Dart y luego el de Flutter, lo que puede ocasionar algún conflicto de archivos. En cambio si escogemos el camino fácil todo está junto en un mismo paquete y mejor organizado. Lo que da la sensación de que es más robusto y estable. Para evitar problemas he escogido el camino fácil. Para instalar Dart/Flutter primero nos dirigimos a la siguiente página web: [Windows install | Flutter](#).

Descargamos el archivo .zip estable (contiene Dart y Flutter):

1. Download the following installation bundle to get the latest stable release of the Flutter SDK:

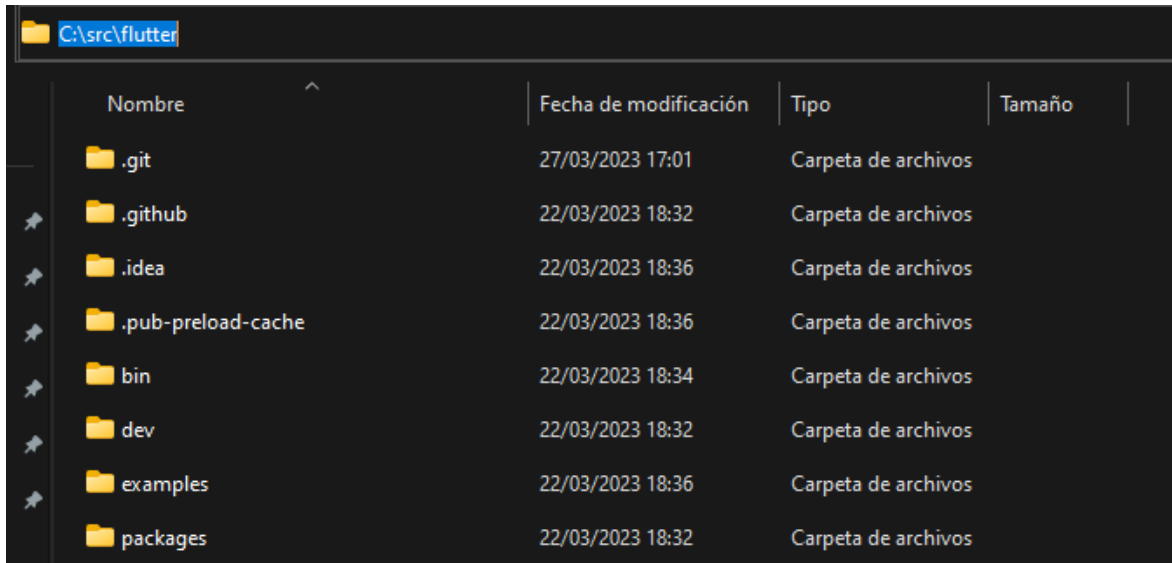
flutter_windows_3.7.10-stable.zip

Si seguimos la documentación nos dice claramente que no debemos alojar esta carpeta en directorios como `C:\Program Files\` ya que requieren niveles de privilegio especiales. O tampoco alojarla en un path que contenga caracteres especiales o espacios.

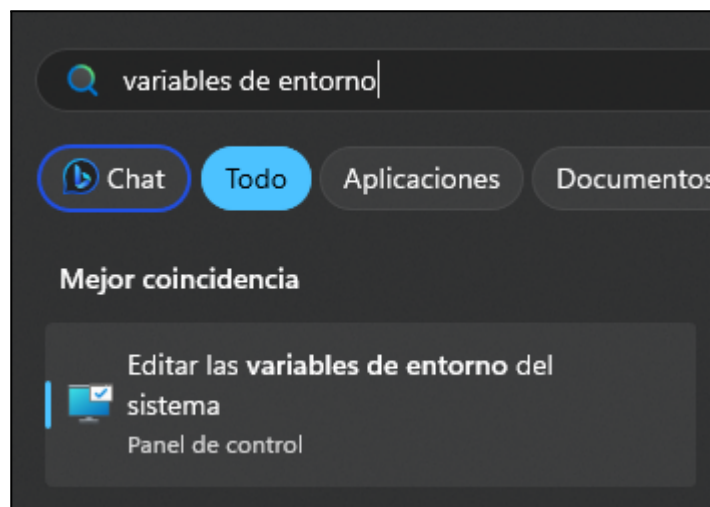
⚠ Warning: Do not install Flutter to a path that contains special characters or spaces.

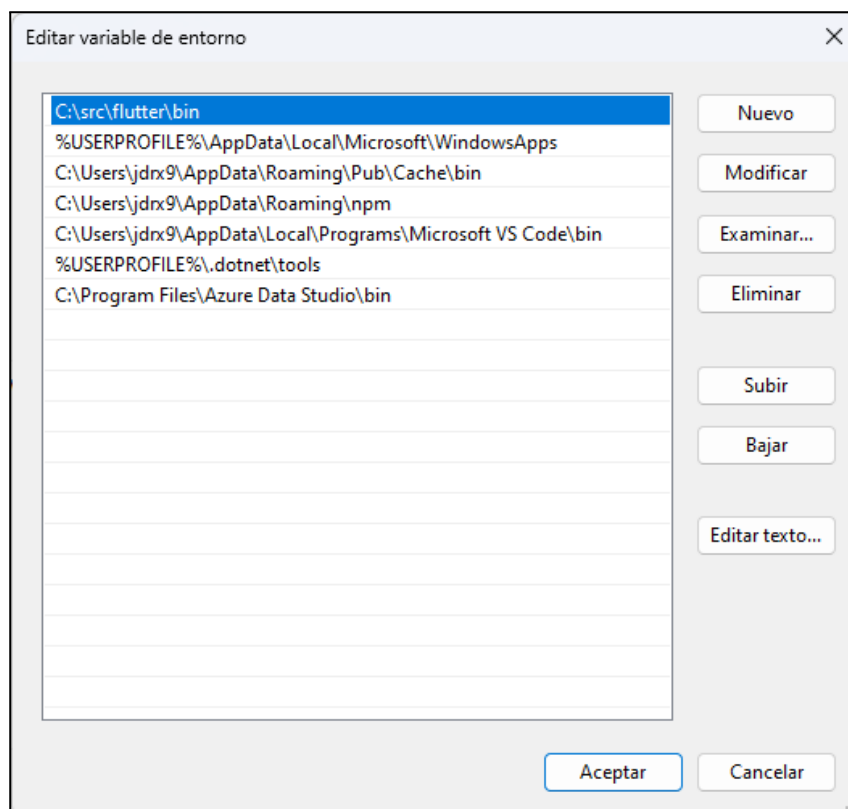
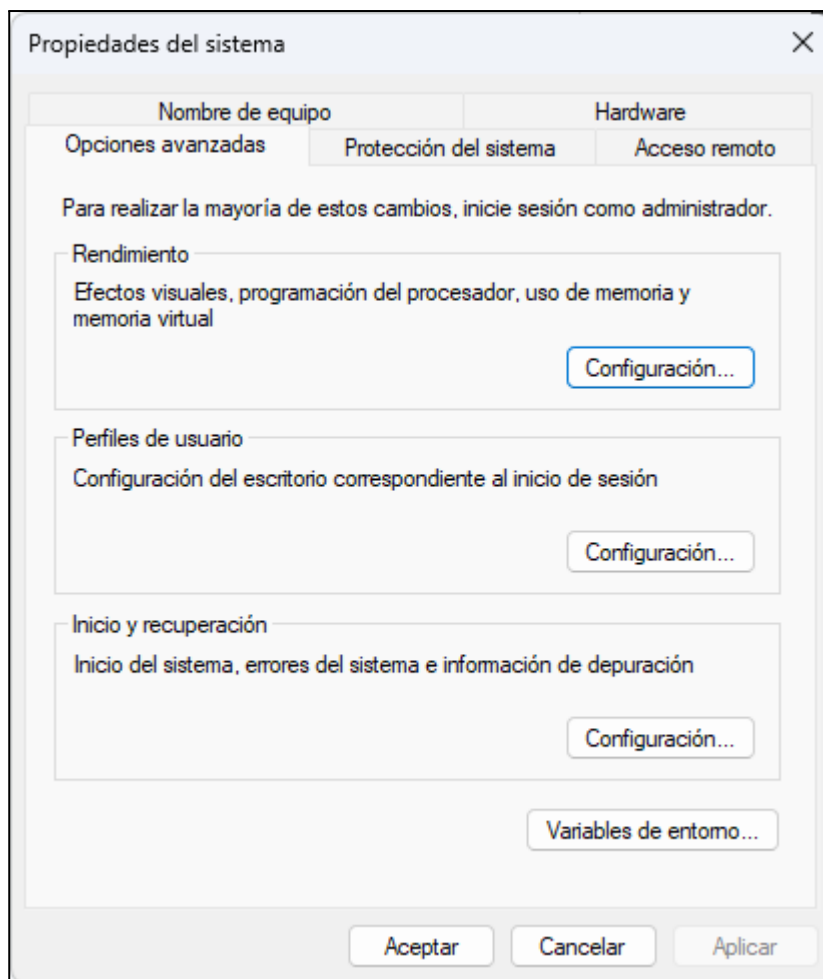
⚠ Warning: Do not install Flutter in a directory like `C:\Program Files\` that requires elevated privileges.

Así que la ruta que nos recomienda es crearnos en C:\ una carpeta llamada 'src' y dentro de ella la carpeta extraída del zip anterior. De tal manera que quede así.



Ahora tenemos que actualizar nuestras variables de entorno. Para ello en Windows no dirigimos a 'Editar variables de entorno' > Variables de entorno... > Variables de usuario > Path





Añadimos la ruta bin y la posicionamos en el primer puesto para darle más prioridad. Si todo ha ido bien al abrir la terminal y preguntarle dónde está alojado Dart y Flutter en nuestro sistema nos debería devolver algo como esto:

```
PowerShell 7.3.3
PS C:\Users\jdrx9> where.exe dart flutter
C:\src\flutter\bin\dart
C:\src\flutter\bin\dart.bat
C:\src\flutter\bin\flutter
C:\src\flutter\bin\flutter.bat
```

Por último debemos ejecutar 'flutter doctor'. Esto nos da información de los componentes extra que tenemos instalados o no para desarrollar en diferentes plataformas. No es obligatorio instalar todos los componentes pero sí recomendable. Por ejemplo si solo vamos a trabajar con web, nos pide que tengamos Google Chrome instalado en nuestro equipo. Para aplicaciones de escritorio Windows, Visual Studio con sus respectivos paquetes. Para aplicaciones Android, Android Studio y así sucesivamente. Si ejecutamos el comando nos aparece lo siguiente:

```
PS C:\Users\jdrx9> flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel stable, 3.7.8, on Microsoft Windows [Version 10.0.22621.1413], locale es-ES)
[x] Windows Version (Unable to confirm if installed Windows version is 10 or greater)
[✓] Android toolchain - develop for Android devices (Android SDK version 33.0.2)
[x] Chrome - develop for the web (Cannot find Chrome executable at .\Google\Chrome\Application\chrome.exe)
    ! Cannot find Chrome. Try setting CHROME_EXECUTABLE to a Chrome executable.
[✓] Visual Studio - develop for Windows (Visual Studio Build Tools 2019 16.11.25)
[✓] Android Studio (version 2022.1)
[✓] VS Code (version 1.77.1)
[✓] Connected device (2 available)
[✓] HTTP Host Availability

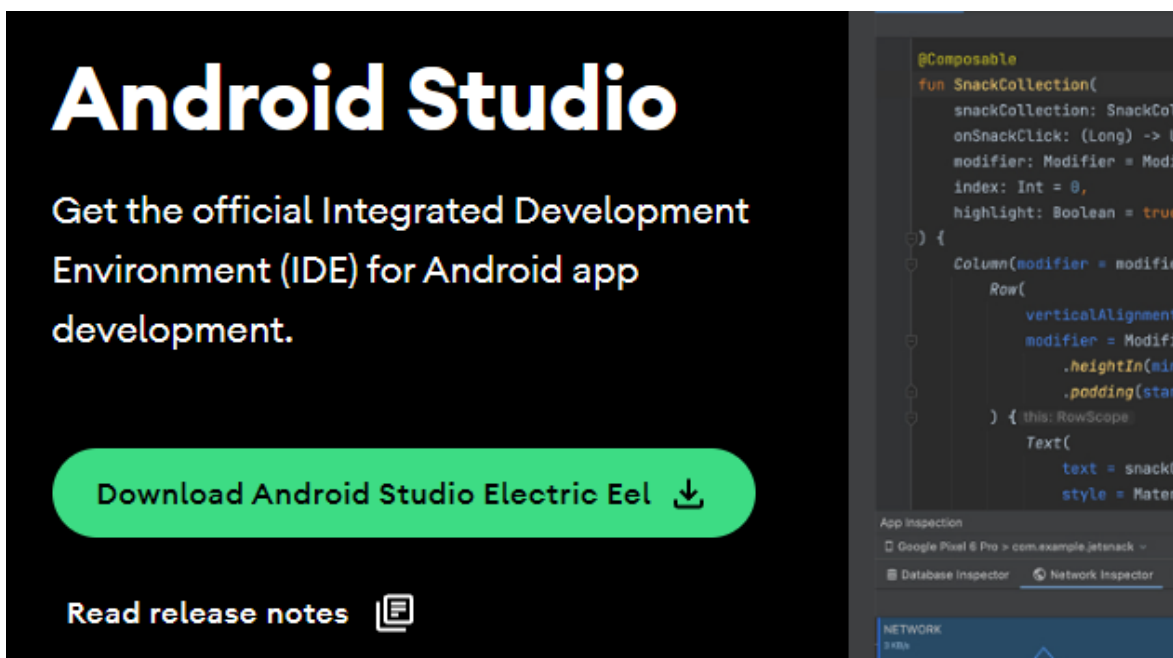
! Doctor found issues in 2 categories.
```

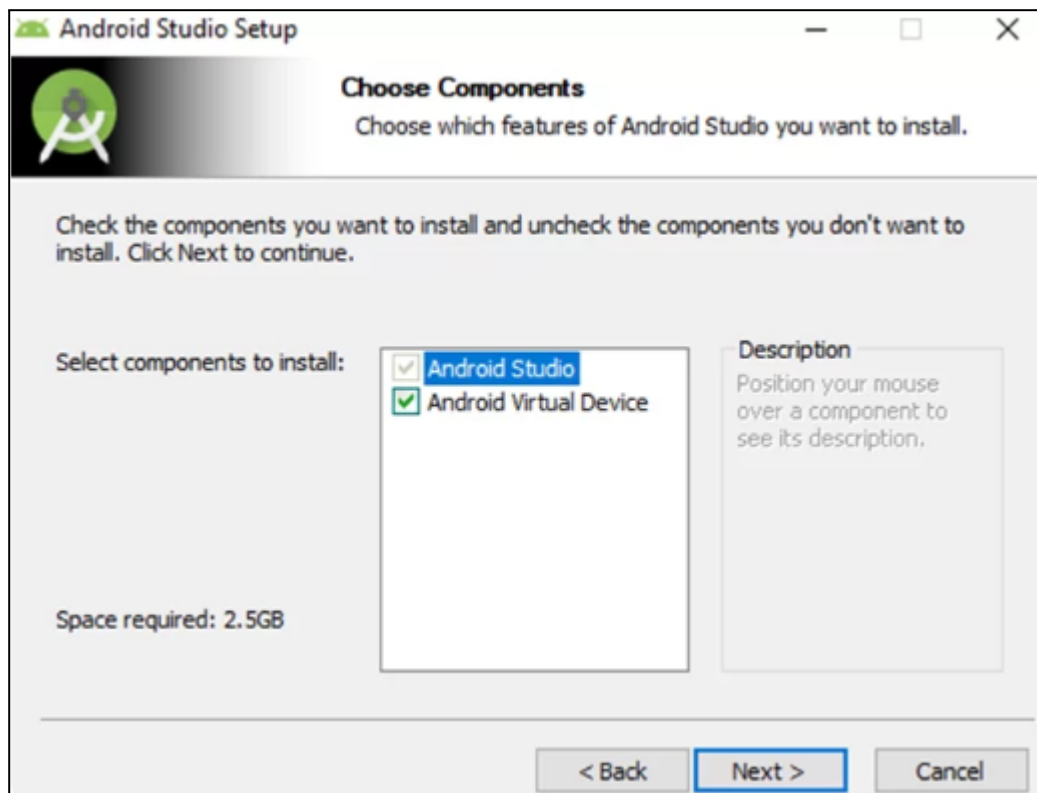
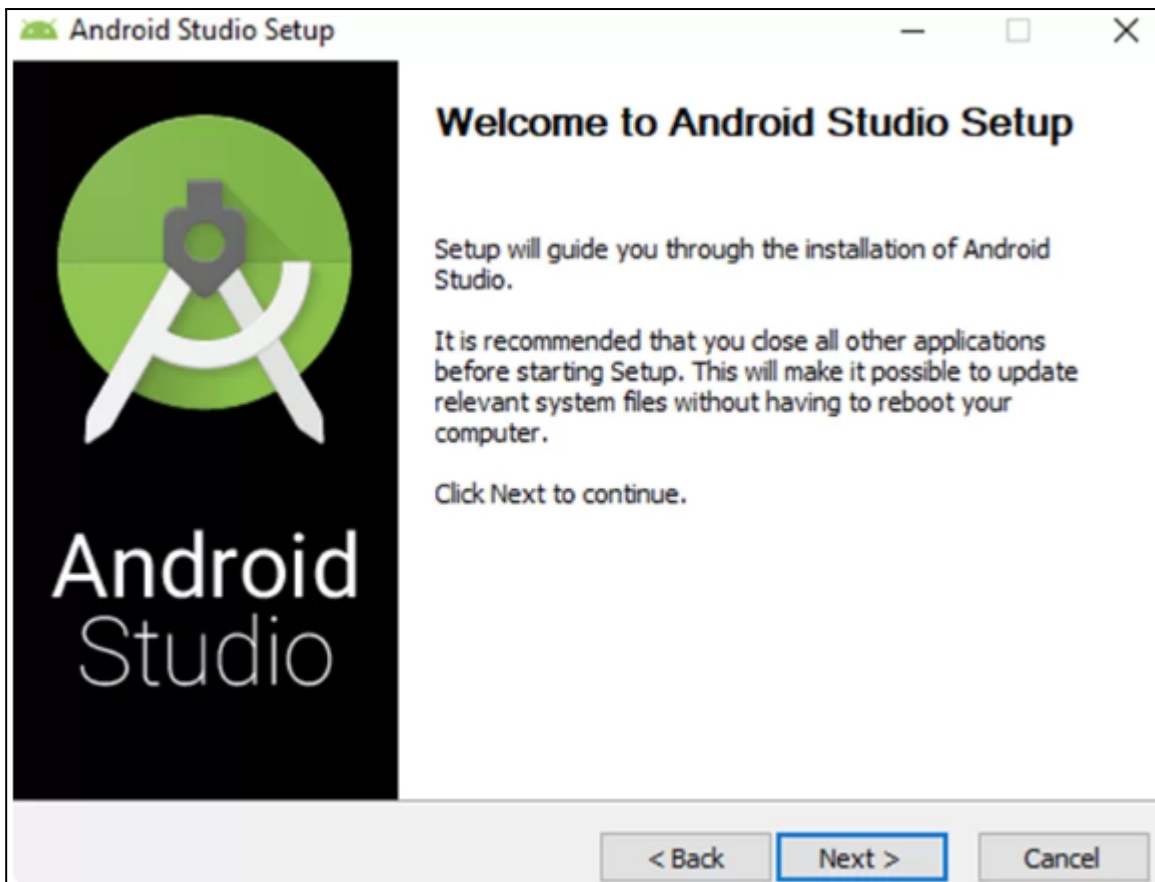
Con esto ya se tendría Dart y Flutter listo para utilizar.

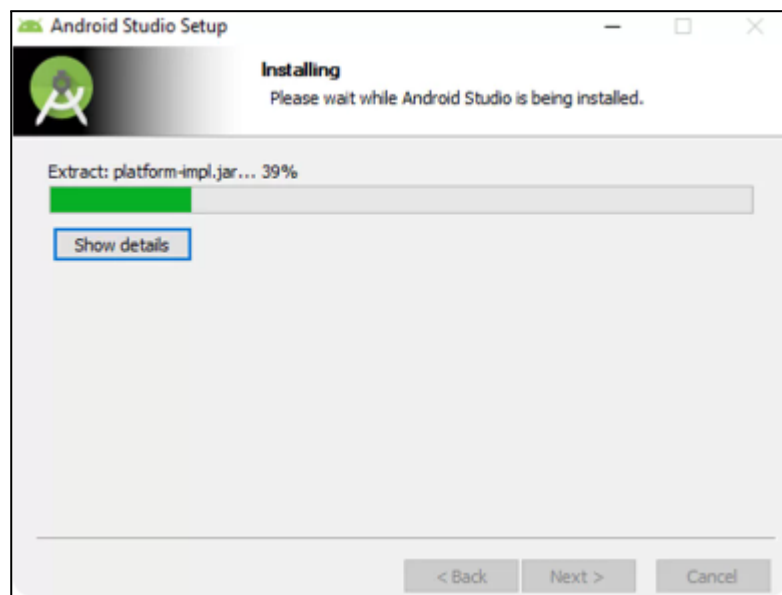
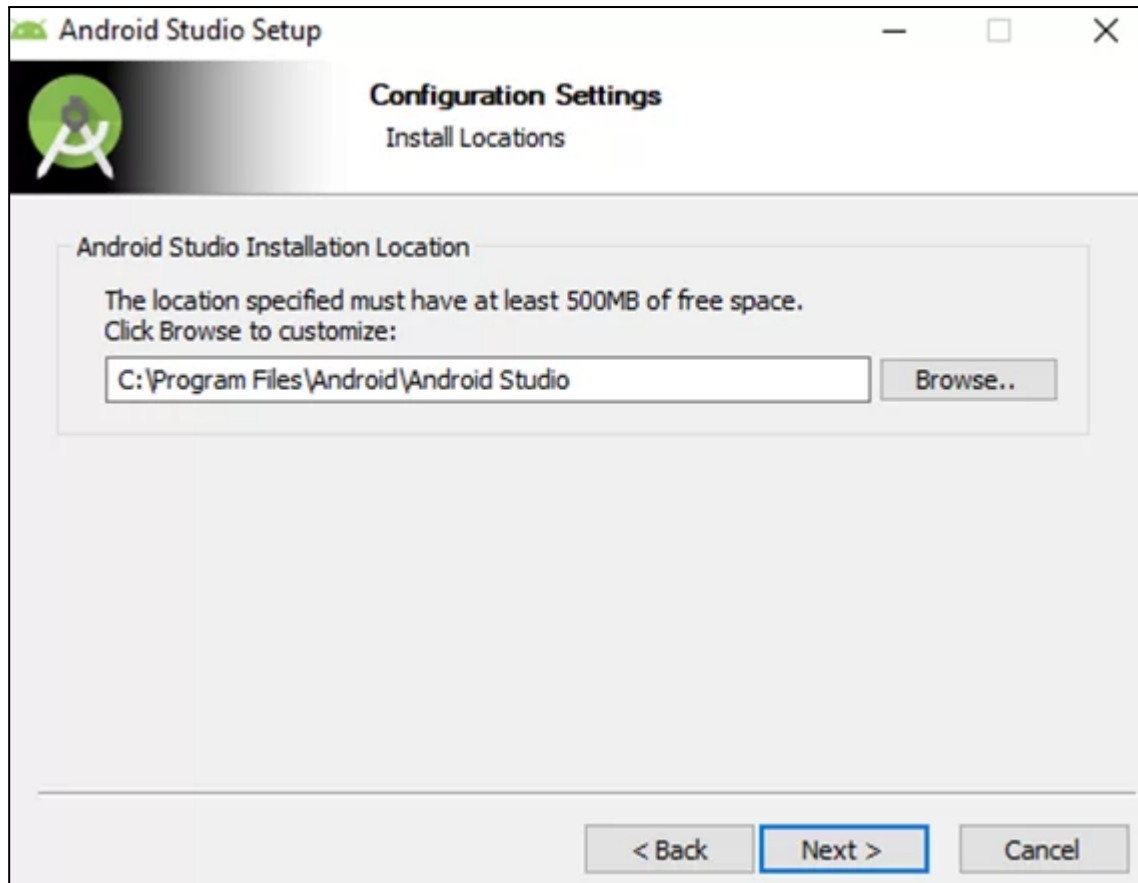
Instalación de un emulador Android

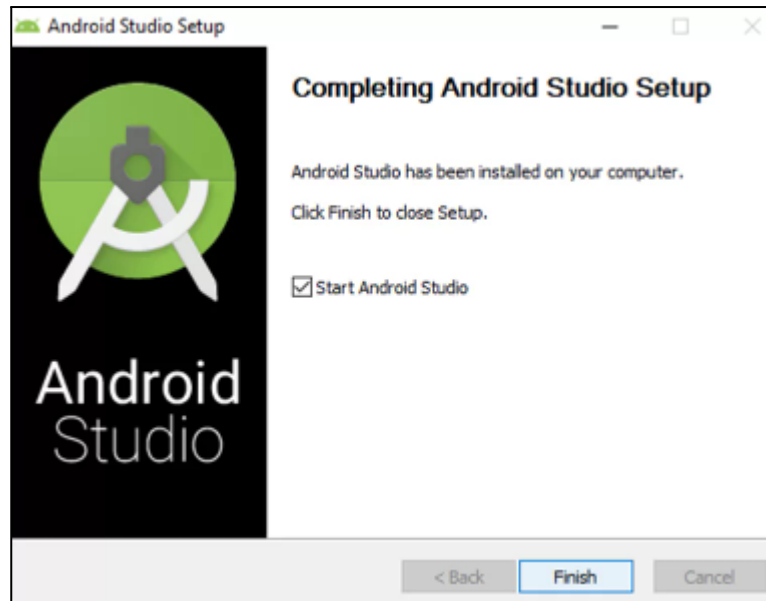
Para poder probar la aplicación mientras se desarrolla en Visual Studio Code tenemos dos opciones. O conectar nuestro móvil Android mediante USB Debug activando el modo desarrollador o creando un emulador. En este proyecto se utilizarán las dos, pero mientras programo prefiero utilizar el emulador y cuando ya tenga casi terminada la App, pasarlo al móvil real y realizar las modificaciones y permisos oportunos para utilizar la aplicación. El uso de un emulador requiere un hardware potente para que sea fluido. En mi caso el emulador funciona sobre un i5 de generación 11 y 16GB de Ram y no he tenido ningún problema de fluidez.

Lo siguiente que hay que hacer para instalar el emulador es descargar Android Studio desde la siguiente página web: [Download Android Studio & App Tools - Android Developers](#)

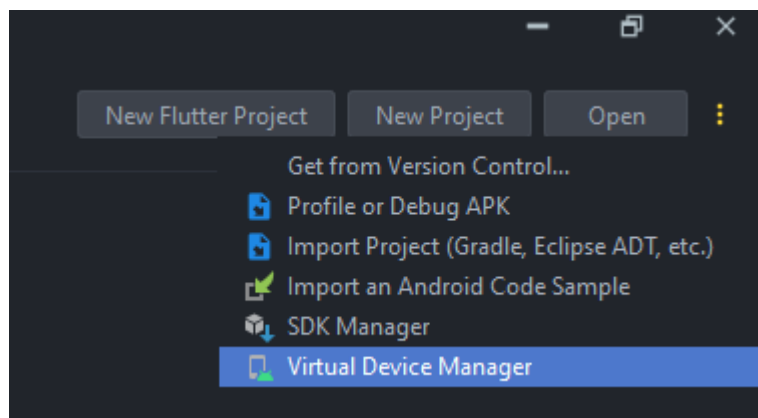


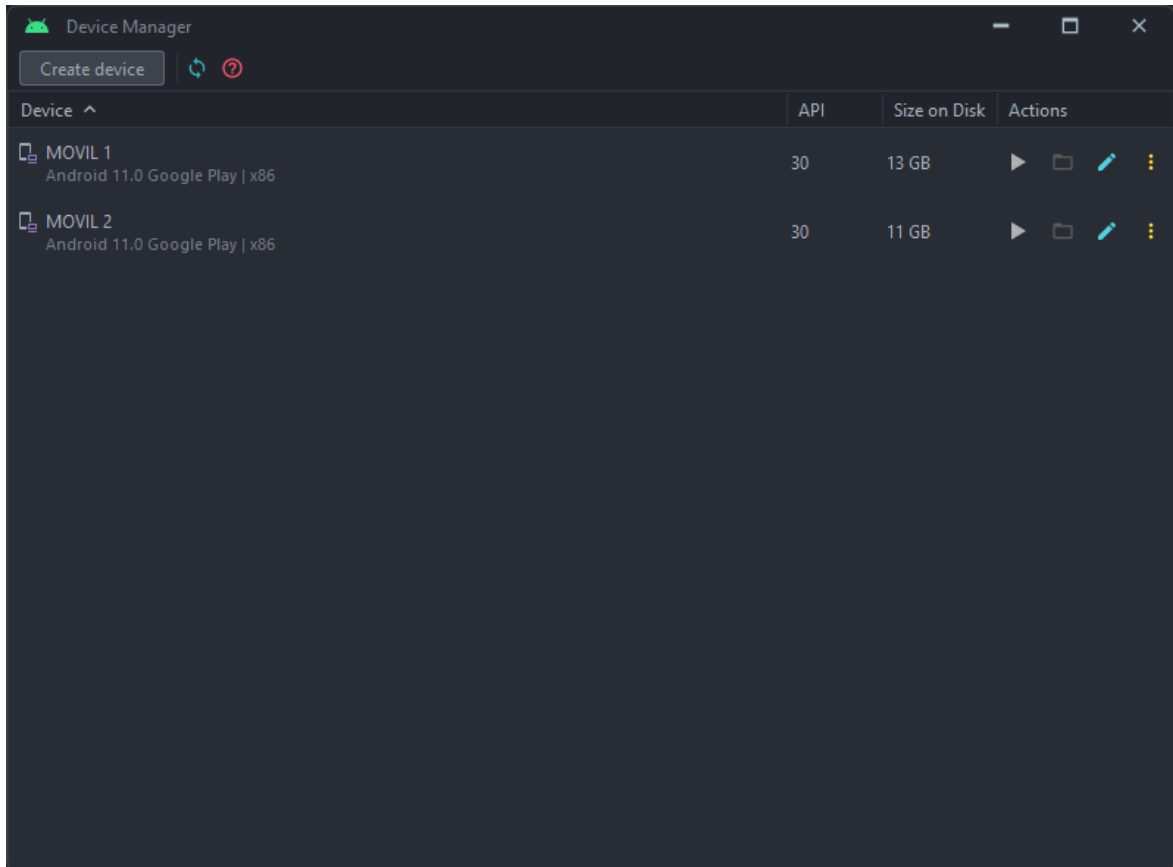




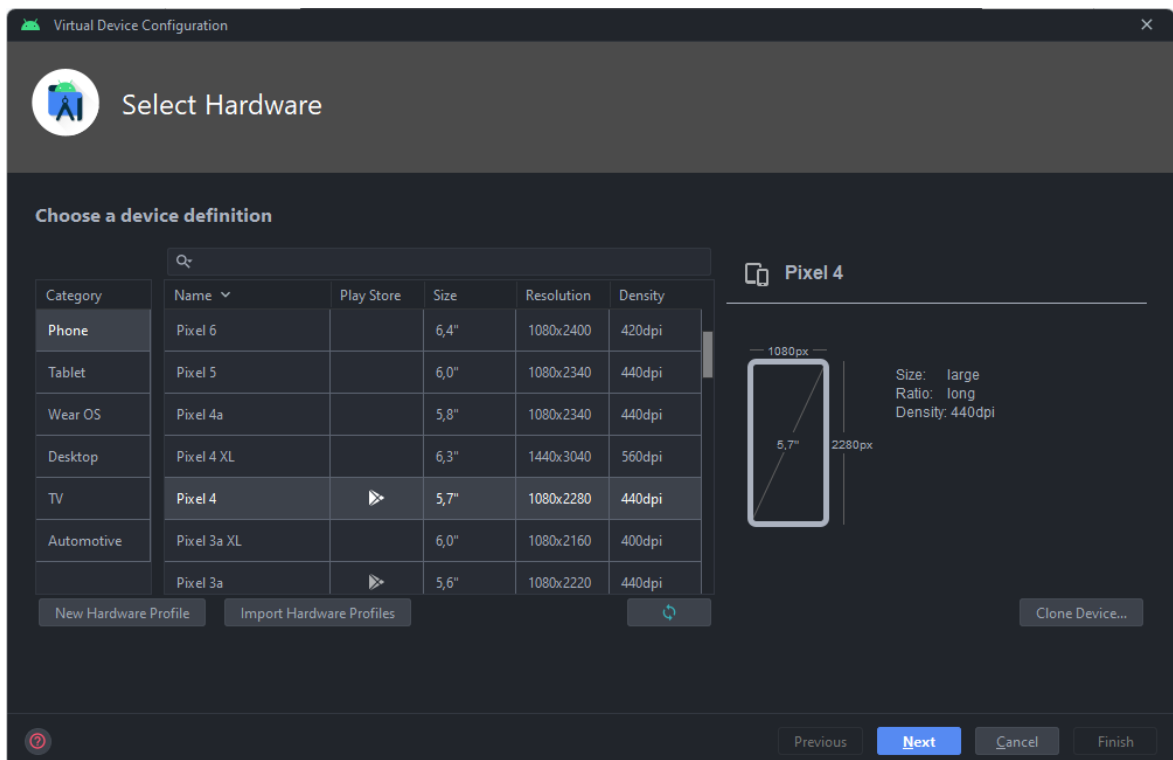


Una vez instalado podremos instalar un emulador de la siguiente manera. Para ello hay que abrir el Android Studio y en la parte superior derecha hacer clic en los tres botones y clicar en Virtual Device Manager:

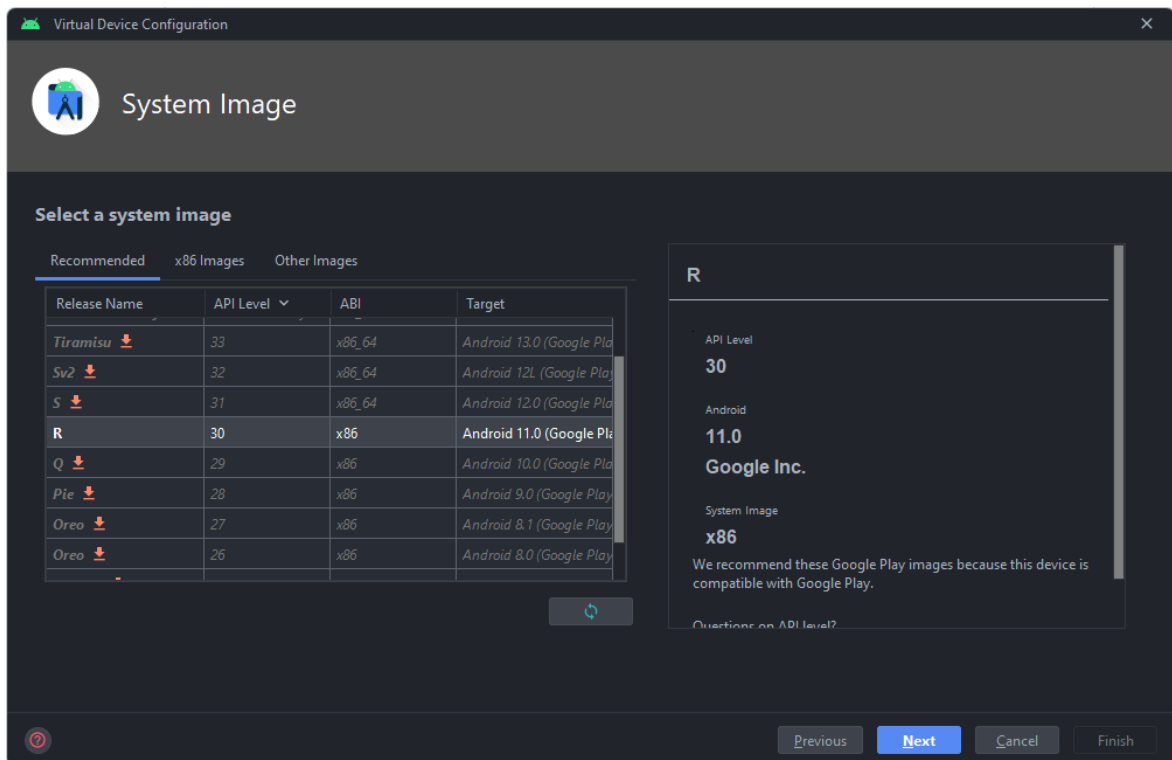




En mi caso ya existen dos emuladores que he creado anteriormente. Pero voy a mostrar el proceso de instalación de un emulador, para ello vamos a Create Device y aparece lo siguiente:

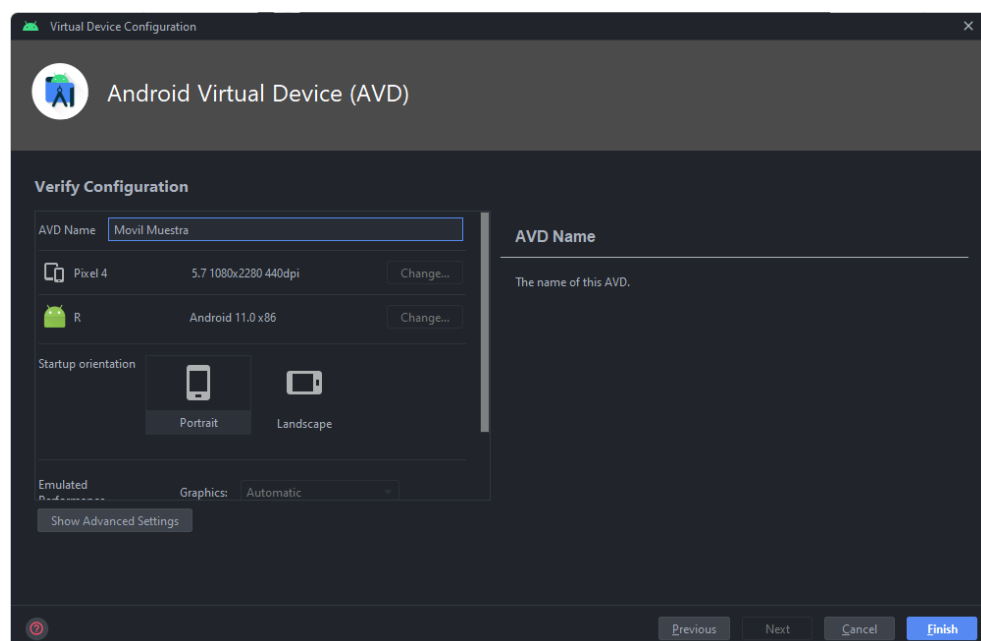


Escogemos un modelo de móvil, en mi caso Pixel 4.



En esta pantalla hay que escoger la imagen Android. Yo recomiendo escoger una API intermedia para equipos normales/medio. Si el hardware que tenemos no es muy potente entonces escogeremos una API de menor nivel, pero también puede perder compatibilidad. En este caso API 30 R. No he tenido ningún problema con él mientras desarrollaba.

Con esto hemos finalizado la instalación de un emulador Android.



Aquí nos muestra un resumen de lo que hemos escogido antes de crearlo. Podemos cambiar el nombre si nos interesa. Cuando estemos listos pulsamos Finish. Cuando lo cree, el móvil será detectado automáticamente en VS Code cuando estemos dentro de un proyecto Flutter para poder ejecutar la aplicación en él.

Acceso al Servidor Amazon AWS

Para el servidor que será el encargado de trabajar por detrás con MongoDB y Flask Socket.IO. Antes he tenido que crear una cuenta de Amazon AWS y crear una nueva instancia de servidor con Ubuntu y MongoDB. Voy a explicar el proceso de instalación de MongoDB en un Servidor AWS Linux y cómo acceder a este mismo servidor desde un equipo local.

Instalación de MongoDB en una distribución Ubuntu

Para realizar esta instalación, la propia página web oficial nos ofrece la documentación con sus pasos a realizar: [Install MongoDB Community Edition on Ubuntu — MongoDB Manual](#). Está disponible para Linux, Redhat, Debian, Mac OS X, Windows, etc.

Lo primero que debemos tener en Linux son permisos de administrador o root. A continuación hay que verificar en qué distribución nos estamos manejando. Para saberlo hay que ejecutar el siguiente comando:

```
grep ^NAME /etc/*release
```

Dependiendo de la distribución tendremos que ir a la documentación correspondiente. En mi caso el resultado es el siguiente:

```
root@ip-172-31-28-9:~# grep ^NAME /etc/*release
/etc/os-release:NAME="Ubuntu"
root@ip-172-31-28-9:~# |
```

Añadimos una GPG Key que nos suministra un repositorio que nos hará falta para instalar posteriormente 'gnupg'.

```
curl -fsSL https://pgp.mongodb.com/server-6.0.asc | \
sudo gpg -o /usr/share/keyrings/mongodb-server-6.0.gpg \
--dearmor
```

```
sudo apt-get install gnupg
```


Lo siguiente es comprobar qué versión estamos utilizando. Para ello se puede ejecutar el siguiente comando: `lsb_release -dc` que me muestra lo siguiente:

```
root@ip-172-31-28-9:~# lsb_release -dc
Description:    Ubuntu 20.04.6 LTS
Codename:       focal
root@ip-172-31-28-9:~#
```

Y ejecutamos lo siguiente:

```
echo "deb [ arch=amd64,arm64 signed-by=/usr/share/keyrings/mongodb-server-6.0.gpg ]
https://repo.mongodb.org/apt/ubuntu focal/mongodb-org/6.0 multiverse" | sudo tee
/etc/apt/sources.list.d/mongodb-org-6.0.list
```

Y recargamos con: **`sudo apt-get update`**

Ya tenemos los componentes necesarios para instalar por fin MongoDB. Para ello:

`sudo apt-get install -y mongodb-org`

Cuando termine ya tendríamos MongoDB instalado en nuestro servidor. Pero para trabajar con él hace falta que iniciemos su servicio que correrá en segundo plano en el servidor. Para iniciar MongoDB debemos ejecutar lo siguiente:

`sudo systemctl start mongod`

Para comprobar su status: **`sudo systemctl status mongod`**

Si ejecutamos '**mongosh**' entraríamos a la Shell de Mongo y obtendremos el connection String necesario para conectarnos desde nuestro equipo local al servidor.

Creando una comunicación entre Flutter y Flask Socket IO

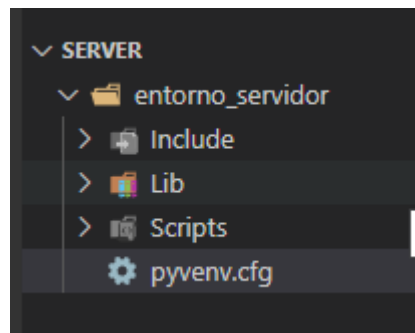
En este apartado voy a explicar como se ha conseguido una conexión entre Flutter y el Servidor Python en Flask Socket IO. Más adelante explicaré cómo se inserta este documento JSON a la base de datos MongoDB.

La parte del servidor (localhost)

Lo primero que se va a programar será el servidor para después poder conectarnos con el cliente a él. Es buena costumbre crear un entorno virtual para distintos proyectos en los que trabajamos con Python. Así que he creado uno siguiendo dos sencillos pasos:

```
python -m venv entorno_servidor
```

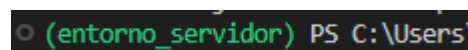
Esto crea una carpeta con los archivos necesarios para el entorno en la que queda estructurada de la siguiente manera:



Para activar nuestro entorno lo haremos desde la consola escribiendo lo siguiente:

```
.\entorno_servidor\Scripts\activate
```

Cuando lo hagamos es fácil saber que estamos dentro de él porque te lo indica de la siguiente manera:

A screenshot of a terminal window showing the prompt '(entorno_servidor) PS C:\Users\' in green text, indicating that the virtual environment is active.

Crear un entorno permite mayor organización entre los paquetes PIP. Habrá proyectos que nos interese tener ciertos paquetes y en otros no. Por ejemplo, si en este entorno del servidor instalamos Flask, solo estará disponible para este entorno y no generará conflicto con los demás entornos. Es una manera limpia de organizar nuestros proyectos.

Ahora con esto podemos empezar a programar. Para ello he creado un nuevo archivo .py que contendrá la inicialización del servidor Flask, incluido sus funciones que se exponen más adelante que puedan escuchar desde el cliente para poder realizar diferentes acciones.

```
# Clave aleatoria
__SECRET_KEY = str(Fernet.generate_key())

# Inicialización de Flask
app = Flask(__name__)

# Configuración
app.config['SECRET_KEY'] = __SECRET_KEY

# Crear el servidor Flask-Socket IO
socketio = SocketIO(app)
```

La parte del cliente

Para realizar la conexión con Flask Socket IO necesitamos depender de un paquete disponible para Flutter en pub.dev llamado:

socket_io_client ([socket_io_client](#) | [Dart Package \(pub.dev\)](#))

Esta librería utiliza los mismos métodos o muy parecidos a Python. Ya que es un port de la librería Node.js de JavaScript que se puede encontrar en GitHub: [socketio/socket.io-client: Realtime application framework \(client\) \(github.com\)](#)

Para ello hay que instalar el paquete en nuestro proyecto de Flutter. Con lanzar el siguiente comando sería suficiente:

flutter pub add socket_io_client

Esta acción añade automáticamente la librería al archivo de configuración 'pubspec.yaml'. Sólo faltaría importar en nuestro archivo esta librería como IO:

```
import 'package:chat/widgets/login.dart';
import 'package:flutter/material.dart';
import 'package:logger/logger.dart';
import 'package:socket_io_client/socket_io_client.dart' as IO;
```

El import se ha colocado en el archivo main.dart, porque lo primero que quiero que haga cuando se abra la aplicación sea establecer la comunicación con el servidor.

Después declaramos un logger para que nos mantenga informado en las partes críticas como es la conexión, desconexión u otro tipos de errores o en donde veamos oportuno.

```
var logger = Logger(printer: PrettyPrinter());
// Se declara en 'late' porque se instanciará más tarde
late io.Socket socket;
```

Dentro de la clase `_MyAppState` se ha declarado una función que nos informa de estos estados y se ha inicializado en `initState()` porque nos interesa que se conecte al Socket en el primer estado de nuestra aplicación.

```
class _MyAppState extends State<MyApp> {
  // Diferentes estados del Socket
  _connectSocket() {
    socket.onConnect((data) => logger.i('Conexión establecida.'));
    socket.onConnectError((data) => logger.e('Error de conexión: $data'));
    socket.onDisconnect((data) => logger.w('Socket.IO desconectado.'));
  }

  @override
  void initState() {
    super.initState();
    // Esta IP nos permite conectarnos al servidor
    socket = io.io("http://10.0.2.2:5000",
      | io.OptionBuilder().setTransports(['websocket']).build());
    _connectSocket();
  }
}
```

Esta dirección IP se utiliza para acceder a los servicios que se ejecutan en la máquina anfitriona desde un emulador o dispositivo virtual. Y es la que se utilizará para desarrollar la aplicación hasta que se realice el deploy final. Es importante mencionar que tipo de transporte queremos utilizar para comunicarnos con el Socket. El método `OptionBuilder()` nos da varias opciones como la conexión automática o manual entre otras. Si no le especificamos el transporte, la conexión no se realizará correctamente.

Para comprobar que realizar la primera conexión primero se va a lanzar el servidor Python:

```
INFO:root:=== Debug Start ===
* Serving Flask app 'ServerIO'
* Debug mode: on
INFO:werkzeug:WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
INFO:werkzeug:Press CTRL+C to quit
INFO:werkzeug: * Restarting with stat
INFO:root:=== Debug Start ===
WARNING:werkzeug: * Debugger is active!
INFO:werkzeug: * Debugger PIN: 590-467-433
```

Por defecto el servidor de Flask si no lo cambiamos nosotros lanza el servidor en localhost sobre el puerto 5000. Ahora si ejecutamos el cliente la conexión se ha realizado correctamente:

```
#0 main (package:chat/main.dart:12:10)
#1 _runMain.<anonymous closure> (dart:ui/hooks.dart:131:23)

💡 Building App...

#0 _MyAppState._connectSocket.<anonymous closure> (package:chat/main.dart:26:39)
#1 EventEmitter.emit.<anonymous closure> (package:socket_io_common/src/util/event_emitter.dart:61:14)

💡 Conexión establecida.
```

Creando una conexión a MongoDB desde Flask

Ahora que el cliente ya se conecta a nuestro servidor, vamos a hacer que el servidor se conecte a una base de datos NoSQL que será MongoDB. Python nos facilitará el trabajo ya que con pocas líneas de código podremos tener una conexión.

Creación de la BBDD

Para empezar se creará un nuevo archivo a parte del servidor que contendrá los datos de conexión y una nueva instancia a MongoDB para después importar esta instancia desde el archivo del servidor. Lo primero que necesitaremos será el String de conexión que obtendremos desde el servidor SSH donde está corriendo MongoDB escribiendo 'mongosh':

```
Current Mongosh Log ID: 644a1be712ed716a8709d332
Connecting to:      mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+1.8.0
Using MongoDB:      6.0.5
Using Mongosh:      1.8.0

For mongosh info see: https://docs.mongodb.com/mongosh-shell/
```

La conexión se puede resolver con tres líneas de código como se muestra a continuación:

```
from pymongo import MongoClient

def get_database():
    # Se almacena la connection String
    CONNECTION_STRING = "mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+1.8.0"

    # Creación de conexión
    client = MongoClient(CONNECTION_STRING)

    # Creación de la BBDD
    return client['Roomfy']

if __name__ == "__main__":
    # Get the database
    dbname = get_database()
```

Importar instancia a Flask y creación de colecciones

Nos dirigimos al archivo donde tenemos la aplicación Flask e importamos esta instancia:

```
from pymongo_get_database import get_database
```

Y se crean las colecciones de la siguiente manera:

```
dbname = get_database()  
collection_usuarios = dbname["usuarios"]  
collection_mensajes = dbname["mensajes"]
```

La colección usuarios almacenará todos los usuarios que se registren en la aplicación y la colección de los mensajes almacenará los mensajes encriptados del chat. Hasta que no le importemos algún documento BSON la colección así como la base de datos no será creada. Con este código sería suficiente para realizar el siguiente paso que es el registro de un usuario.

Realizando el registro de un usuario

Para explicar el registro me centraré en la parte únicamente del Backend, comenzando desde la parte en la que el usuario presiona el botón de 'Registro'. Para el registro de usuario se han de rellenar los campos Nombre y Usuario, ambos conforman el usuario con el que se deberá iniciar sesión. Estos campos tienen validación, únicamente acepta caracteres en minúsculas y números. Nada de mayúsculas o caracteres especiales. Para ello se debe crear una GlobalKey que valide estos campos.

```
final _formKeySignUp = GlobalKey<FormState>();
```

Este `_formKeySignUp` tiene un método llamado `Validate()` que devuelve un booleano en caso de que los campos cumplen con las condiciones que se han programado. Si se cumple se realizará el proceso de registro:

```
onPressed: () async {
  User nuevoUsuario = User(
    name: nombreController.text,
    username: usuarioController.text); // User

  String jsonString = jsonEncode(nuevoUsuario);

  if (_formKeySignUp.currentState!.validate()) {
    await register(jsonString);
    await isRegister();
  }
},
child: const Text('Registrarme'),
```

Cuando el usuario presione el botón de registro se generará un nuevo `User()`. Este `User` es un modelo que está en la carpeta 'models' del proyecto y contiene lo siguiente:

```
class User {
  final String name;
  final String username;

  User({required this.name, required this.username});

  factory User.fromJson(Map<String, dynamic> json) {
    return User(
      name: json['name'],
      username: json['username'],
    );
  }

  Map<String, dynamic> toJson() => {'name': name, 'username': username};
}
```

La siguiente línea simplemente codifica el usuario para que se pueda enviar al servidor. Y después como hemos dicho anteriormente se valida con un `if` si el formulario es correcto, si es así se comienza la comunicación. Para ello hay dos funciones asíncronas.

`register(String json)`

```
Future<void> register(String json) async {
  socket.emit('registro', json);
}
```

Esta función emite un evento al servidor con emit, emitiendo el nombre del evento 'registro' y los datos que en este caso es el json codificado anteriormente. Cuando este evento se envía el servidor está continuamente escuchando este evento que debe llamarse igual:

```
@socketio.on('registro')
def handle_new_user(new_user):
    # Decodificar JSON
    nuevo_usuario = json.loads(new_user)

    # Verificar si el usuario ya existe
    usuario_existente = collection_usuarios.find_one(nuevo_usuario)

    if not usuario_existente:
        results = collection_usuarios.insert_one(nuevo_usuario)

        if results.acknowledged:
            logging.info(f'Se ha registrado un nuevo usuario: {nuevo_usuario}')
            socketio.emit('registrado', 0)
    else:
        logging.info(f'Ya esta registrado este usuario: {nuevo_usuario}')
        socketio.emit('registrado', 1)
```

Esta función se encarga primero de decodificar el JSON recibido, verificar si el usuario existe. realiza una serie de comprobaciones. Si el usuario que está pidiendo el registro no existe en la base de datos de MongoDB se inserta y se comprueba que la grabación de datos ha sido correcta. Si es correcta se vuelve a emitir al cliente un nuevo evento 'registrado' y un 0 para indicar que todo ha ido bien. En caso contrario se envía el mismo evento pero un 1. Aquí entró en juego la siguiente función del cliente.

isRegister()

Cada función le acompaña un await porque nos interesa que primero verifique si el usuario existe y hasta que no acabe no empezar con la siguiente comprobación. Esta función escucha un evento diferente en este caso porque solo queremos que lo escuche una vez. Por lo tanto se ha utilizado el siguiente método socket.once():

```
Future<void> isRegister() async {
    socket.once('registrado', (data) {
```


Socket.once escucha el evento solo una vez que recibe desde el servidor y se obtiene en data la respuesta del servidor, que en nuestro caso almacenará un 0 o un 1. Teniendo esto en cuenta ya podemos realizar comprobaciones y crear diferentes Widgets dependiendo de la respuesta del servidor. En nuestro caso se va a crear un AlertDialog() que informa si el usuario existe o se ha registrado correctamente.

```
if (estaRegistrado == 0) {  
  return showDialog(  
    context: context,  
    builder: (BuildContext context) {  
      return AlertDialog(  
        title: const Text("¡Bien!"),  
        content: const Text("Te has registrado con éxito!"),  
        actions: [  
          TextButton(  
            child: const Text("Iniciar sesión"),  
            onPressed: () {  
              Navigator.of(context, rootNavigator: true).pop();  
            },  
          ), // TextButton  
        ],  
      ); // AlertDialog  
    },  
  );  
}
```

Realizando el login de usuario

Para realizar el login de usuario se han seguido los mismos pasos que en el proceso de registro. Lo único que cambia es el nombre del evento y la lógica en la parte del servidor que es la siguiente:

```
@socketio.on('login')
def login(user):
    # Decodificar JSON
    usuario_registrado = json.loads(user)

    # Verificar si el usuario está en la BBDD
    usuario_existe = collection_usuarios.find_one(usuario_registrado)

    if usuario_existe:
        logging.info(f'Inicio de sesión: {usuario_registrado}')
        socketio.emit('logeado', 0)
    else:
        logging.info(f'Error de login: {usuario_registrado}')
        socketio.emit('logeado', 1)
```

Ésta función realiza menos acciones, ahora solo decodifica el usuario que se ha introducido en la pantalla de Login. Verifica que el usuario existe en la base de datos y si es así emite otro evento 'logeado' con un 0 de que todo ha ido bien y en caso contrario un 1. Ahora en la parte de Flutter se comprueba los datos que nos llegan del servidor. Si es un 0 entra en la aplicación y sino vuelve a mostrar un AlertDialog() informando al usuario de que ya existe y poder introducir los datos de nuevo.

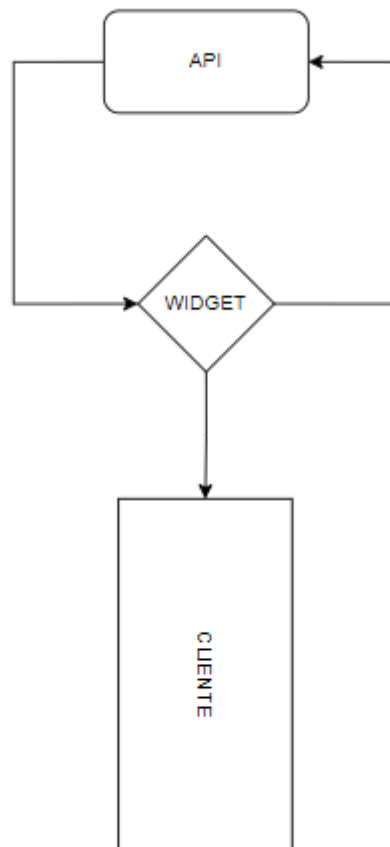
```
if (estaLogin == 0) {
    String username = '${nombreController.text}@${nombreController.text}';
    Navigator.pushReplacement(
        context,
        MaterialPageRoute(builder: (context) => Chat(username: username)),
    );
} else if (estaLogin == 1) {
    return showDialog(
        context: context,
        builder: (BuildContext context) {
            return AlertDialog(
                title: const Text("Vaya ... "),
                content: const Text("Este usuario no existe."),
                actions: [
                    TextButton(
                        child: const Text("Probar de nuevo"),
                        onPressed: () {
                            Navigator.of(context, rootNavigator: true).pop();
                        },
                    ), // TextButton
                ],
            ); // AlertDialog
        },
    );
}
```

Enviando y recibiendo mensajes

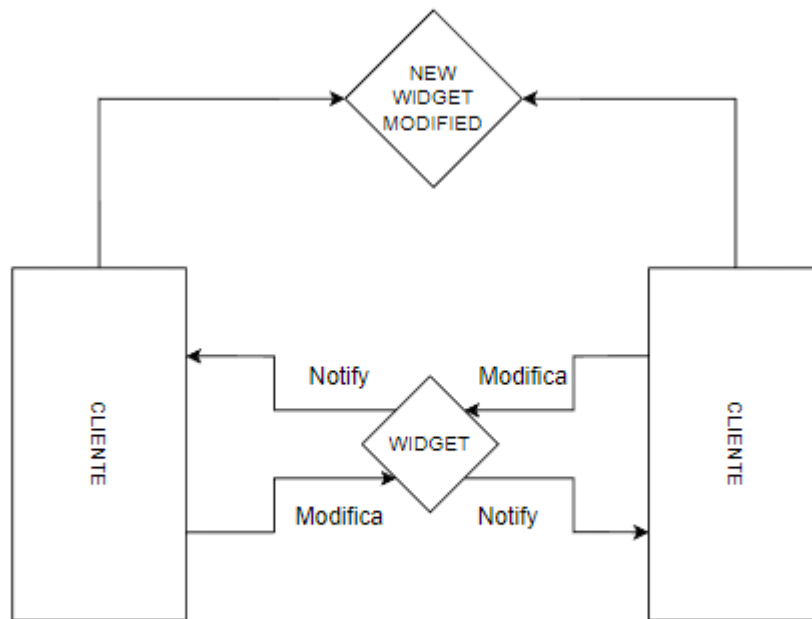
En este apartado vamos a realizar la parte principal de la aplicación que es el envío y recibo de mensajes a cada cliente conectado en grupo. Este apartado será transversal y el más amplio porque va a recorrer los tres componentes que hemos ido programando hasta ahora, empezando por el envío de un mensaje desde el cliente, llegando al servidor para posteriormente guardar el mensaje en la base de datos. Un viaje de ida y vuelta entre estos componentes.

Providers

Como bien dice el título, la clave de todo reside en los providers. En otras ocasiones hemos hecho proyectos con Widgets estáticos, cambiando mínimamente el valor por ejemplo de un Card Widget conectado a una API. Por ejemplo:



Los providers van algo más allá y nos permite mandar un mensaje a todos los Listeners o escuchantes de ese mismo Widget de que se ha producido una modificación en nuestro Widget. Cuando este cliente o cualquier otro emite una notificación (`NotifyListeners()`) el Widget actualiza su estado mostrando su nuevo contenido para todos. Esto es clave a la hora de realizar un ListView, una lista de mensajes de chat.



Esquema general de un provider

Teniendo esto en cuenta vamos a proceder a la programación. Para ello se ha añadido una nueva carpeta al proyecto llamada Providers. Aquí vamos a almacenar un archivo llamado chat.dart. Pero antes, debemos instalar la librería providers que no viene por defecto en Flutter. Esto lo haremos desde el repositorio oficial de librerías pub.dev ejecutando en la terminal el siguiente comando:

flutter pub add provider

Una vez hecho esto, el archivo chat.dart contiene lo siguiente:

```

import 'package:chat/models/message.dart';
import 'package:flutter/material.dart';
import 'package:flutter/foundation.dart';

class ChatProvider extends ChangeNotifier {
  final List<Message> _messages = [];

  List<Message> get messages => _messages;

  addNewMessage(Message message) async {
    _messages.add(message);
    notifyListeners();
  }
}

```

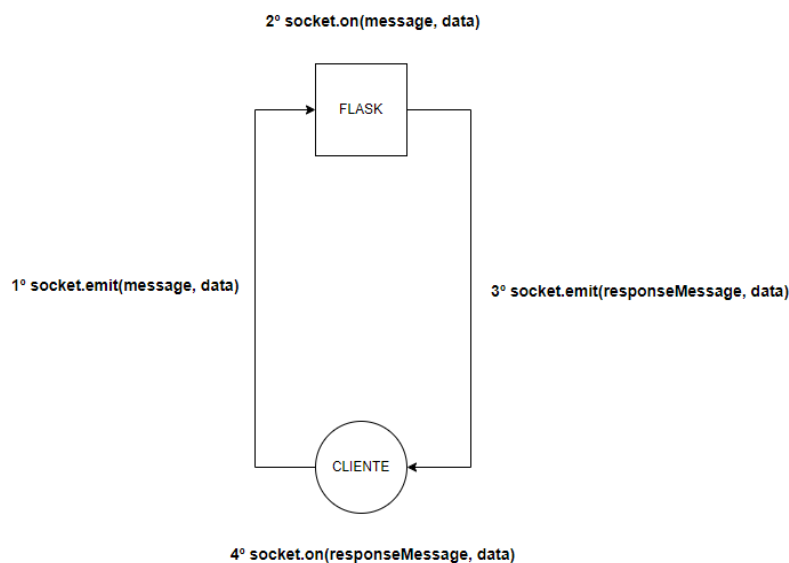
Empecemos por los imports. Primero se importa el modelo Message, que básicamente es el mensaje en sí. Lo siguiente es un import esencial y básico que introduce de forma automática Flutter para indicarnos que estamos trabajando sobre diseño Material 3 de Android. Y por último el paquete Foundation nos permite hacer uso de notifyListeners().

Se crea una clase ChatProvider que extiende ChangeNotifier. Se crea una lista mutable donde será recogido una lista de estos mensajes. Y por último una función para añadir un nuevo mensaje para posteriormente, y parte más importante, avisar a los demás que están escuchando de que se ha añadido un nuevo mensaje.

Hecho esto, vamos a la pantalla de chat. Como antes, solo me voy a centrar en las funciones backend, no en el diseño. Voy a aportar el código y un esquema del camino que realiza, creo que de esta forma se verá mejor que una explicación extensa. Para empezar hemos creado ésta función:

```
_sendMessage() {
  String jsonMessage = _messageInputController.text.trim();
  socket.emit(
    'message',
    jsonEncode(
      {'message': jsonMessage, 'senderUsername': widget.username}));
  _messageInputController.clear();
}
```

El InputController corresponde al teclado donde recogemos el texto y lo limpiamos con trim() para guardarlo en un variable, que será la que enviaremos al servidor mediante el socket.emit que hay a continuación. Para finalizar limpiamos el texto del mensaje.



Ciclo de vida cliente/servidor de un mensaje

```
@socketio.on('message')
def listen_message(message):
    # Decodificar JSON
    new_message = json.loads(message)
    logging.debug(new_message)

    # Enviar respuesta al cliente
    socketio.emit('responseMessage', new_message)
```

```
_responseMessage() {
  socket.on(
    'responseMessage',
    (data) => Provider.of<ChatProvider>(context, listen: false)
      .addNewMessage(Message.fromJson((data))));
}
```

Esta última función `_responseMessage()` es muy importante, porque se dedica a almacenar los mensajes con el Provider a esa lista de mensajes que hemos mencionado antes. Por lo tanto cada vez que tengamos una respuesta por parte del servidor, que está escuchando continuamente gracias al `socket.on()` guardará un mensaje. Pero, ¿cómo actualizar una lista de mensajes para que los demás podamos ver estos mensajes?. Para ello hay que generar un Consumer, que como el nombre indica será el encargado de consumir el Provider y los cambios. Y dentro de este Consumer se anida un `ListView` donde irán almacenados los mensajes. El código es algo extenso, si se quiere ver en detalle os animo a verlo dentro del archivo `Chat.dart`, dentro de `Screens` [Línea 69].

```
body: Column(children: [
  Expanded(
    child: Consumer<ChatProvider>(
      builder: (_, provider, __) => ListView.separated(
```

Para finalizar y que todas las funciones explicadas anteriormente funcionen, es necesario aplicarlas en el `initState` de nuestra screen. `_getMessagesFromDb()` es una función que veremos ahora.

```
@override
void initState() {
  super.initState();
  _getMessagesFromDb();
  _responseMessage();
}
```

Persistencia de chat

Hasta ahora podíamos comunicarnos con todos los conectados, pero cuando salíamos y volvíamos a entrar todos los mensajes se borraban. Esto era debido a que en el proceso de comunicación con el servidor, el servidor no guardaba en la base de datos los mensajes. Vamos a explicar cómo guardar los mensajes en MongoDB y por último cómo obtener estos mensajes para que cuando se inicie la aplicación se muestren los mensajes que se enviaron anteriormente.

Guardando mensajes en MongoDB

Para ello hay que añadir y modificar algunas funciones de la parte del servidor. Empezaremos por la siguiente:

```
@socketio.on('message')
def listen_message(message):
    # Decodificar JSON
    new_message = json.loads(message)
    logging.debug(new_message)

    # Enviar respuesta al cliente
    socketio.emit('responseMessage', new_message)

    collection_mensajes.insert_one(new_message)
```

Ahora después de dar una respuesta al cliente se le ha ordenado que inserte el mensaje en la colección de mensajes. Ahora cada vez que generamos un nuevo Message desde el cliente lo guardará en la base de datos.

Recuperando mensajes desde MongoDB

En esta parte debemos añadir código nuevo tanto en la parte del cliente como en el servidor. Vamos a empezar por el cliente:

```
_getMessagesFromDb() {
    // Emitimos una orden al Server de que queremos obtener datos de MongoDB
    socket.emit('getMsgFromDb');
    // Escuchamos su respuesta una vez, nos devuelve una lista de mensajes
    // y los añade a la lista de mensajes del provider
    socket.once('getMsgs', (data) {
        List<dynamic> oldData = data;
        for (var msg in oldData) {
            Provider.of<ChatProvider>(context, listen: false)
                .addNewMessage(Message.fromJson(msg));
            logger.i(msg['message']);
        }
    });
}
```

La parte del servidor se programó lo siguiente acorde con el cliente:

```
@socketio.on('getMsgFromDb')
def getMessagesFromMongoDB():
    # Declaramos una lista que almacena los mensajes de Mongo
    data = []
    # Recorremos la colección de mensajes para añadirla a lista
    for msg in collection_mensajes.find({}, {'message': 1, 'senderUsername': 1, '_id': 0}):
        logging.info(msg['message'])
        data.append(msg)
    # Emitimos al cliente nuestra respuesta y le enviamos esta lista
    socketio.emit('getMsgs', data)
    logging.info(data)
```

Para finalizar es importante en la parte del cliente añadir al initState() de que queremos obtener primero los mensajes desde la base de datos, para ello añadimos la función que hemos creado anteriormente de esta manera:

```
@override
void initState() {
    super.initState();
    _getMessagesFromDb();
    _responseMessage();
}
```


Deploy

Proceso de build final de la aplicación y publicación disponible para el usuario final.

Cambios en la parte del servidor

Hasta ahora se ha estado desarrollando sobre el entorno local. Esta parte explica cómo subir nuestro código a nuestro servidor SSH para que nos podamos conectar desde el móvil al servidor. Primero se va a crear un nuevo entorno virtual Python en el servidor para alojar los dos archivos de nuestro proyecto ServerIO y get_database. El proceso de creación de un entorno virtual Python es exactamente igual que he explicado al principio así que me lo voy a saltar. Vamos a copiar nuestros dos archivos mencionados anteriormente al servidor SSH, para ello ejecutamos dos comandos, uno para cada archivo:

```
scp .\ServerIO.py root@13.37.121.111:/ruta_entorno_virtual
scp .\pymongo_get_database.py root@13.37.121.111:/ruta_entorno_virtual
```

Con esto ya tendríamos nuestros archivos en el servidor:

```
root@ip-172-31-28-9:/home/server# ls -l
total 16
-rw-r--r-- 1 root root 2224 May  7 17:39 ServerIO.py
drwxr-xr-x 2 root root 4096 May  7 17:41 __pycache__
drwxr-xr-x 6 root root 4096 May  7 17:35 app
-rw-r--r-- 1 root root  438 May  7 17:39 pymongo_get_database.py
root@ip-172-31-28-9:/home/server# |
```

Vamos a decirle ahora a ServerIO que cuando ejecutemos Flask escuche cualquier dirección, para ello escribimos **nano ServerIO.py** y **modificamos lo siguiente:**

```
if __name__ == '__main__':
    logging.info('=== Debug Start ===')
    socketio.run(app, debug=True, host='0.0.0.0')
```

Cambios en la parte del cliente

Los únicos cambios que hay que realizar en la parte del cliente es en el archivo Main.dart donde creamos la conexión al servidor. La IP va a cambiar ya que debemos poner la del servidor SSH:

```
@override
void initState() {
  super.initState();
  // Para dev: "http://10.0.2.2:5000"
  // Para deploy: "http://13.37.121.111:5000"
  socket = io.io("http://10.0.2.2:5000",
    | io.OptionBuilder().setTransports(['websocket']).build());
  _connectSocket();
}
```

Build y APK

Por defecto el SO Android ejecuta sus aplicación finales con extensión .apk Para generar la build de la aplicación completa y obtener esta APK para que lo puedan disfrutar otros usuarios, tendremos que ejecutar en la terminal de nuestro proyecto la siguiente línea de código:

```
flutter build apk
```

El proceso de compilación dependerá del tamaño de nuestro proyecto y ordenador. Una vez que ha terminado el proceso, el archivo se guarda en la siguiente ruta para su distribución:

```
build\app\outputs\flutter-apk
```

[Descarga aquí Roomfy](#)

Conclusiones

En términos generales no me han surgido graves problemas, salvo en las partes críticas. Como son las líneas de código de la conexión entre Flutter y Socket. Y el deploy en el servidor SSH. A través de prueba y error y documentarme conseguí sacarlo adelante. La pantalla de chat tuve que mirar también la documentación sobre los Providers y algún vídeo de YouTube donde explicaba su funcionamiento. Para la documentación técnica de Flutter, Mongo o Python, en vez de perder tiempo buscando por sus páginas preguntaba al copiloto de Microsoft Bing donde me ayudó bastante a solucionar muchas dudas que me iban surgiendo. Pero estas búsquedas solo eran con la finalidad de buscar entre la documentación y no bloques de código. También he aprendido bastante sobre servidores, ya que es un tema que no estaba muy involucrado y no he llegado a profundizar tanto. Gracias a este proyecto he aprendido cómo manejarme con un servidor SSH, copiar archivos, crear entornos virtuales python en Linux y levantar procesos en segundo plano para que los dispositivos se puedan conectar a él.

La parte positiva de esta aplicación es que ofrece una alternativa de comunicación a las que conocemos actualmente, obviamente con mucha menos implementación y opciones ya que solo ofrece una comunicación básica. Pero se podría escalar e ir implementando nuevas características como chat individual, mensajes encriptados, respuestas a un solo mensaje, perfil de usuario, etc... Como parte negativa al realizarse sobre una minimal API si los usuarios a futuro son bastantes, seguramente se tenga que cambiar esta arquitectura por otro Framework más grande.

En cuanto a código la variable socket que se declara en el Main.dart podría haberse creado en un singleton ya que se utiliza la misma variable de conexión durante toda la vida de la aplicación. Para poder llamarla desde diferentes Widgets y tenerlo más organizado. Así como el logger que se declara en cada pantalla. Estas son las principales mejoras que realizaría.

Bibliografía

Flask-SocketIO — Flask-SocketIO documentation. (s. f.).

<https://flask-socketio.readthedocs.io/en/latest/>

Flutter documentation. (s. f.). Flutter.

<https://docs.flutter.dev/>

Flutter – Material Design 3. (s. f.). Material Design.

<https://m3.material.io/develop/flutter>

El tutorial de Python. (s. f.). Python documentation.

<https://docs.python.org/es/3/tutorial/index.html>

Welcome to Python.org. (2023, 13 abril). Python.org.

<https://www.python.org/>

Install MongoDB Community Edition on Amazon Linux — MongoDB Manual. (s. f.).

<https://www.mongodb.com/docs/manual/tutorial/install-mongodb-on-amazon/>

MongoDB. (s. f.). *Build A Python Database With MongoDB | MongoDB.*

<https://www.mongodb.com/languages/python>

Visual Studio Code - Code Editing. Redefined. (2021, 3 noviembre).

<https://code.visualstudio.com/>