

# Proyecto de Fin de Grado



DAM  
Curso 2022-2023

## Integrantes

Luz María Lozano Asimbaya  
Víctor Manuel Monzón Pérez

# Índice.

---

1. Introducción.....	4
2. Módulos formativos aplicados en el trabajo.....	5
3. Herramientas y lenguajes utilizados.....	7
1. Dart.....	7
2. Flutter.....	8
3. Python.....	10
3.1 Flask.....	10
4. Firebase.....	11
5. GIT / GITHUB.....	13
4. Componentes del equipo y aportación realizada por cada estudiante.....	14
5. Fases del proyecto.....	16
5.1 ¿CÓMO FUNCIONA LUDI VERBORUM?.....	16
5.2 PLANTEAMIENTO APLICACIÓN.....	17
5.3 DESCRIPCIÓN Y ARQUITECTURA DEL PROYECTO.....	20
5.4 DESARROLLO.....	21
5.5 BBDD: FIREBASE.....	22
5.5.1 Authentication.....	23
5.6 SERVER.....	24
5.6.1 SCRAPER.....	24
5.6.2 DAO.....	27
5.6.3 FLASK.....	28
5.7 CLIENTE.....	30
5.8 APP MOBILE: FLUTTER.....	32
5.8.1 Creación pantallas con Flutter.....	34
5.9 SCREENS PRINCIPALES.....	40

5.9.1 Pantalla de Juego. ....	40
5.9.2 Pantalla Opciones. ....	43
5.9.3 Pantalla Diccionario. ....	44
5.10 DIAGRAMA DE CASOS DE USO. ....	47
5.11 DIAGRAMA DE PANTALLAS. ....	48
6. Conclusiones y mejoras. ....	49
6.1 Conclusión. ....	49
6.2 Mejoras. ....	49
7. Bibliografía. ....	51

## I. Introducción.

---

Ludi Verborum, en latín significa juegos de palabras. Esta aplicación es una aplicación tanto para el entretenimiento como para el conocimiento que ayuda al mejor conocimiento de las palabras.

Esta aplicación está conectada directamente con la web de la RAE, de esta manera la información de las palabras y sus definiciones estará siempre actualizada. Este juego está inspirado en el popular juego de televisión llamado “Pasapalabra”, esto es, el juego nos mostrará la inicial y una de las definiciones, la finalidad es que el usuario averigüe cuál es la palabra en un tiempo marcado.

También se puede usar la aplicación a modo de diccionario personal, la aplicación tiene una funcionalidad que permite añadir palabras al diccionario, esta se añadirá con todas las definiciones de la misma. De esta manera podremos ir añadiendo y conociendo las definiciones de las palabras.

El contenido del diccionario se mostrará en una de las pantallas, donde también habrá una vista de las definiciones.

Finalmente se podrá configurar tanto el nombre de usuario como el tiempo de juego, esto último añadirá complejidad al juego.

## 2. Módulos formativos aplicados en el trabajo.

---

### 2.1 Programación.

Se ha usado la lógica de programación aprendida en este módulo con el lenguaje de Java. Estos conocimientos nos han servido para poder usar el lenguaje de programación de Flutter con el que se ha desarrollado esta aplicación.

### 2.2 Programación Multimedia y Dispositivos móviles.

Desarrollo de la aplicación para Android e IOS.

### 2.3 BBDD.

Desarrollo de la base de datos para el almacenamiento de las palabras.

### 2.4 Acceso a Datos.

Extracción de las palabras y de las entradas de la base de datos y utilizarlas en la aplicación.

### 2.5 Desarrollo de Interfaces.

Aplicación de todos los aspectos teóricos para el desarrollo de UX y de GUI.

### 2.6 Programación de Servicios y Procesos.

Vinculación de la app con la base de datos por medio de sockets y llamadas con protocolo HTTP para las peticiones HTML a [dle.rae.es](http://dle.rae.es).

### 2.7 Empresa e Iniciativa Emprendedora.

Branding, Naming y desarrollo de posible estrategia para escalabilidad.

## 2.8 Sistemas de Gestión Empresarial.

Recolección de datos de usuario, posible estrategia CRM.

## 2.9 Entornos de Desarrollo.

De este módulo hemos usado:

- GitHub: para la gestión del repositorio y de las versiones del código.
- Visual Studio Code: para el desarrollo de la aplicación.

## 2.10 Lenguaje de Marcas y Sistemas de Gestión de Información.

Manejo de archivos JSON para gestión de información.

## 2.11 Inglés.

Toda la documentación para el desarrollo de la aplicación está en gran parte en inglés.

### 3. Herramientas y lenguajes utilizados.

---

#### 1. Dart.



Dart es un lenguaje de programación de código abierto desarrollado por Google. Fue creado con el objetivo de proporcionar un enfoque moderno y eficiente para el desarrollo de aplicaciones web y móviles. Dart combina características de lenguajes como C++, Java y JavaScript, ofreciendo a los desarrolladores un conjunto de herramientas versátiles para crear aplicaciones de alto rendimiento.



#### Uso Hot Restart.

Permite probar rápidamente los cambios en su aplicación durante el desarrollo, incluidas las ediciones de código de varios archivos, los recursos y las referencias.

Impulsa los nuevos cambios al paquete de aplicaciones existente en el destino de depuración, lo que da como resultado un ciclo de compilación e implementación mucho más rápido.

#### Uso Hot Reload.

Esta funcionalidad de Flutter te ayuda a rápida y fácilmente experimentar, construir UIs, añadir funcionalidades y arreglar bugs. Hot reload trabaja inyectando ficheros de código fuente actualizados en la VM Dart en ejecución.

#### Uso para muchos propósitos.

Es un lenguaje que se puede usar casi para cualquier cosa, tanto en aplicaciones web como en servidores. También con varios frameworks para aplicaciones de consola o móvil.

#### Similitud con otros lenguajes.

Dart es muy similar a lenguajes como JavaScript, Java y C++, esto hace muy fácil aprender este lenguaje conociendo uno de los lenguajes mencionados. Además, Dart consta de mucho apoyo para la asincronía, por lo que trabajar con generadores e iterables es sencillo.

#### Tiene varias herramientas integradas.

Dart se diseñó con el objetivo de hacer el proceso de desarrollo lo más cómodo y rápido, por esto, tiene un conjunto extenso de herramientas integradas; como su propio gestor de paquetes, compiladores o transpiladores, analizador y formateador. Por otro lado, la máquina virtual de Dart y la compilación Just-In-Time hacen que los cambios realizados en el código se puedan ejecutar inmediatamente.

## 2. Flutter.



Flutter es un framework de código abierto desarrollado por Google para crear aplicaciones nativas de alta calidad para iOS, Android y la web, desde una sola base de código. Utiliza el lenguaje de programación Dart y proporciona un conjunto de herramientas y widgets que facilitan el desarrollo de interfaces de usuario atractivas y fluidas.

Sigue una arquitectura basada en widgets, lo que significa que todo en Flutter es un widget. Los widgets se utilizan para construir la interfaz de usuario y también se utilizan para manejar eventos y actualizaciones de estado. Flutter proporciona diferentes tipos de widgets, como widgets de diseño, widgets interactivos y widgets de bajo nivel, que se combinan para crear la interfaz de usuario deseada.





#### Rapidez de desarrollo.

Flutter permite un desarrollo rápido y eficiente gracias a su hot reload, que permite ver los cambios en tiempo real sin reiniciar la aplicación. Esto acelera el proceso de desarrollo y facilita la corrección de errores.

#### Interfaces atractivas.

Esta funcionalidad de Flutter te ayuda a rápida y fácilmente experimentar, construir UIS, añadir funcionalidades y arreglar bugs. Hot reload trabaja inyectando ficheros de código fuente actualizados en la VM Dart en ejecución.

#### Alto rendimiento.

Es un lenguaje que se puede usar casi para cualquier cosa, tanto en aplicaciones web como en servidores. También con varios frameworks para aplicaciones de consola o móvil.

#### Compatibilidad multiplataforma.

Dart es muy similar a lenguajes como JavaScript, Java y C++, esto hace muy fácil aprender este lenguaje conociendo uno de los lenguajes mencionados. Además, Dart consta de mucho apoyo para la asincronía, por lo que trabajar con generadores e iterables es sencillo.

#### Comunidad activa.

Dart se diseñó con el objetivo de hacer el proceso de desarrollo lo más cómodo y rápido, por esto, tiene un conjunto extenso de herramientas integradas; como su propio gestor de paquetes, compiladores o transpiladores, analizador y formateador. Por otro lado, la máquina virtual de Dart y la compilación Just-In-Time hacen que los cambios realizados en el código se puedan ejecutar inmediatamente.

### 3. Python.



Python es un lenguaje de programación interpretado y de alto nivel. Se destaca por su sintaxis legible y su enfoque en la simplicidad y la facilidad de uso.

Es utilizado en diversos ámbitos, como desarrollo web, análisis de datos, inteligencia artificial, automatización de tareas y mucho más. Su popularidad se debe en parte a su comunidad activa y al amplio soporte que ofrece.

Una de las características distintivas de Python es su enfoque en la legibilidad del código, lo que facilita la colaboración y el mantenimiento de proyectos a largo plazo. Además, Python es un lenguaje versátil que permite tanto programación estructurada como orientada a objetos, y ofrece características como el manejo de excepciones y la recolección automática de basura, que simplifican el desarrollo y mejoran la fiabilidad de las aplicaciones.

#### 3.1 Flask.



Flask es un framework con código en Python, es decir, una colección de librerías y módulos que permite a los desarrolladores crear aplicaciones sin preocuparse por detalles

específicos como protocolo y gestión de hilo, entre otros. Fue desarrollado por Armin Ronacher, quien lideró el equipo internacional de Python, destaca entre otros por su facilidad; no tiene una gran curva de aprendizaje.

Su principal función es la de levantar de forma sencilla servidores ya que está diseñado para facilitar la creación de API RESTful y principalmente aplicaciones web. Proporciona mecanismos para gestionar rutas, manejar solicitudes y respuestas HTTP y la gestión de hilos (threads).

#### 4. Firebase.



Firebase es una potente plataforma de back-end que ofrece un conjunto de herramientas y servicios para ayudar a los desarrolladores a construir, escalar y mantener aplicaciones web y móviles. Proporciona a los desarrolladores características fáciles de usar, como usar bases de datos en tiempo real, autenticación, alojamiento y capacidades de aprendizaje automático. Fue creada en 2011 y adquirido por Google en 2014.



En cuanto a la estructura de los datos, estos se almacenan como objetos JSON, en documentos que se encuentran anidados en colecciones. A diferencia de una base de datos de SQL, en este tipo de bases de datos no hay tablas ni registros, son más bien un árbol de JSON que se alojan en la nube. Cuando se añaden datos al árbol estos son nodos de la estructura con una clave asociada. La estructura sería como en la imagen,

```
{
  "users": {
    "alovelace": {
      "name": "Ada Lovelace",
      "contacts": { "ghopper": true },
    },
    "ghopper": { ... },
    "eclarke": { ... }
  }
}
```

## 5. GIT / GITHUB.



Git es un Sistema de Control de Versiones Distribuido (DVCS) utilizado para guardar diferentes versiones de un archivo (o conjunto de archivos) para que cualquier versión sea recuperable cuando lo desee.

Git permite a los desarrolladores guardar y gestionar diferentes versiones de archivos y proyectos. En lugar de guardar solo la versión más reciente de un archivo, Git guarda un historial completo de todos los cambios realizados en el tiempo. Esto significa que se puede acceder a cualquier versión anterior de un archivo en cualquier momento.

Además de la funcionalidad de control de versiones, Git también ofrece características para facilitar la colaboración entre desarrolladores. Permite a múltiples personas trabajar en un mismo proyecto al mismo tiempo, fusionando sus cambios de manera eficiente. También facilita la identificación de quién hizo cada cambio y permite agregar comentarios explicativos a los cambios realizados.

Git es un sistema distribuido, lo que significa que cada desarrollador tiene una copia completa del repositorio en su propio dispositivo. Esto proporciona una mayor redundancia y seguridad, ya que si un servidor central falla, los desarrolladores aún pueden acceder a las versiones y la historia del proyecto desde sus copias locales.

Por otra parte, GitHub es una plataforma basada en la web donde los usuarios pueden alojar repositorios Git. Facilita compartir y colaborar fácilmente en proyectos con cualquier persona en cualquier momento.

GitHub también fomenta una participación más amplia en proyectos Código Abierto al proporcionar una manera segura de editar archivos en repositorios de otros usuarios.

#### 4. Componentes del equipo y aportación realizada por cada estudiante.

---

Los componentes del grupo son:

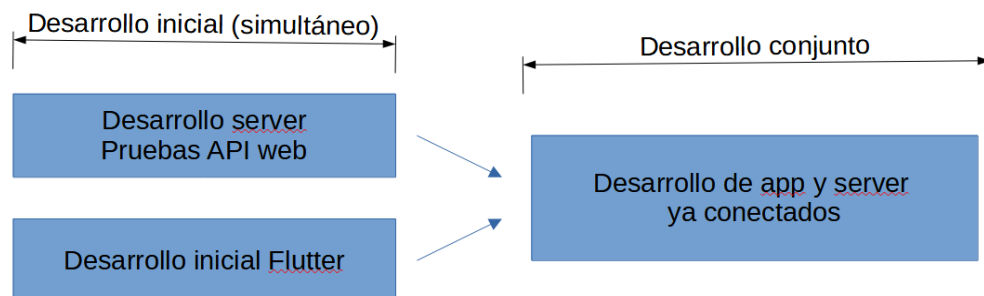
- Victor Manuel Monzón Pérez.
- Luz María Lozano Asimbaya.

Aportaciones de cada estudiante:

En una primera instancia, conjuntamente analizamos la web de la RAE, punto imprescindible para el desarrollo de la aplicación. Posteriormente, buscamos información sobre técnica de web-scraping para ver las respuestas que obteníamos.

Al estructurar el proyecto en Servidor y Cliente pudimos avanzar simultáneamente en el desarrollo, Víctor centrándose más en el servidor (DAO, Flask) y Luz en las funciones de Flutter.

Una vez que tuvimos ambos procesos de desarrollo para que proveyesen una función suficiente, ambos participamos en el desarrollo completo de la aplicación, tanto evolucionando toda la parte correspondiente al Python, como las diferentes pantallas de la app indistintamente, ya que conocíamos y compartíamos los avances para ser más diligentes en el desarrollo de la App gracias a una comunicación fluida y un repositorio Git en el que aportábamos los avances del proyecto y lo aunábamos para estar completamente coordinados.



## LUDI VERBORUM

 By project
  Board
  All Tasks
  Timeline
 

Aa Task name	Status	Assignee	Due	Priority	Tags	Project
✓ Entrega anteproyecto	Done	Luz Lozano Vi_Monzon	April 5, 2023	High		LUDI VERBORUM
✓ Definir qué tecnologías vamos a usar	Done	Luz Lozano Vi_Monzon	April 16, 2023	High		LUDI VERBORUM
✓ Configurar firebase	Done	Luz Lozano Vi_Monzon	April 10, 2023			LUDI VERBORUM
✓ Crear Server	Done	Vi_Monzon	April 20, 2023			LUDI VERBORUM
▼ ✓ Definir las actividades que hay que realizar para la app	Done	Luz Lozano Vi_Monzon	April 19, 2023	High		LUDI VERBORUM
✓ Splash	Done	Luz Lozano	May 10, 2023	Low		LUDI VERBORUM
✓ login	Done	Vi_Monzon	April 30, 2023	Medium		LUDI VERBORUM
✓ pantalla inicial	Done	Luz Lozano	April 29, 2023	High		LUDI VERBORUM
✓ pantalla juego	Done	Vi_Monzon	May 1, 2023	High		LUDI VERBORUM
✓ pantalla añadir palabra	Done	Luz Lozano	May 18, 2023	High		LUDI VERBORUM
✓ pantalla opciones	Done	Vi_Monzon	May 21, 2023	Low		LUDI VERBORUM
✓ pantalla diccionario	Done	Luz Lozano Vi_Monzon	May 22, 2023			LUDI VERBORUM
✓ pantalla puntuacion	Done	Luz Lozano Vi_Monzon	May 25, 2023			LUDI VERBORUM
+ New sub-item						
▶ ✓ Entrega Intermedia	Done	Luz Lozano Vi_Monzon	May 4, 2023	High		LUDI VERBORUM

## 5. Fases del proyecto.

---

### 5.1 ¿CÓMO FUNCIONA LUDI VERBORUM?



#### LUDI VERBORUM

Ludi Verborum es un juego de palabras en el que el usuario tendrá que adivinar palabras en función de su definición y la letra inicial.

Para poder jugar el usuario debe estar logueado, sino tiene cuenta creada, hay que crear una nueva cuenta, una vez creada la cuenta se cargará un diccionario estándar para todos los usuarios, una vez dentro de la aplicación cada usuario podrá añadir palabras a su diccionario y, por ende, en el juego.

La finalidad del juego es adivinar, en función de la letra y la definición que nos muestra, la palabra que corresponde similar al concurso televisivo Pasapalabra.

Por ejemplo,

**Letra 'A'**

*'Pájaro cantor de 15 a 20 cm de largo, con el plumaje pardo con bandas negruzcas en el dorso y blanco ocráceo en la parte inferior; tiene la cola larga, ahorquillada y blanca por los lados, y una cresta corta y redonda; es insectívoro y gregario, y anida en los campos de cereales.'*

**La respuesta sería la palabra 'Alondra'.**



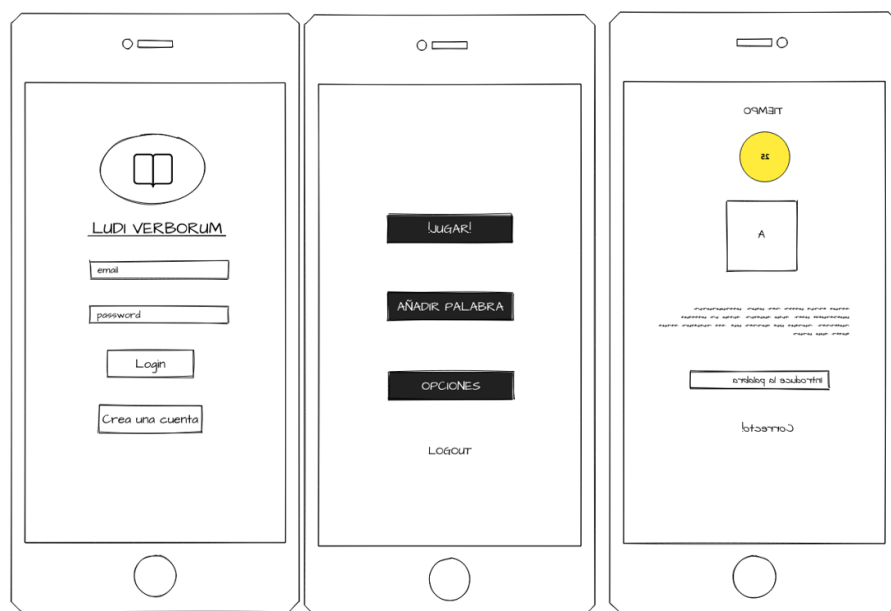
A parte de la función principal del juego, este nos da la posibilidad de añadir y borrar palabras del diccionario, esto modificará el juego para hacerlo más dinámico.

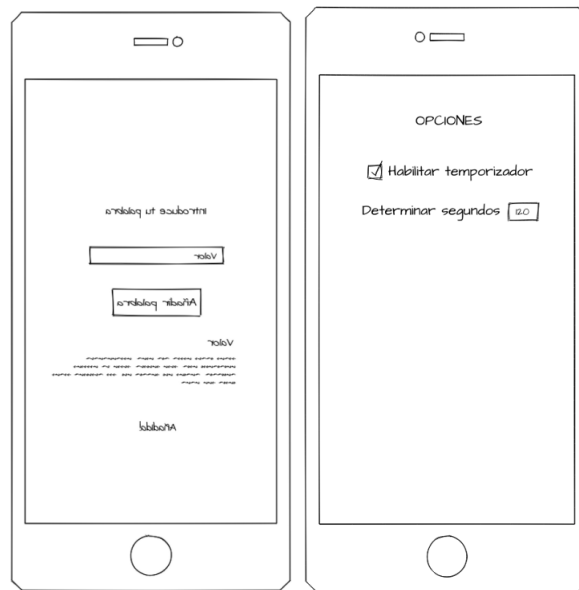
Por último, la aplicación tiene acceso al diccionario que alimenta el juego, de esta manera también se puede usar como diccionario personal, ya que, al añadir la palabra automáticamente se verán todas las definiciones de la palabra.

## 5.2 PLANTEAMIENTO APLICACIÓN.

Antes de empezar con el desarrollo y seleccionar las herramientas con las que trabajar se hizo un primer esquema de las funcionalidades que se quería plantear.

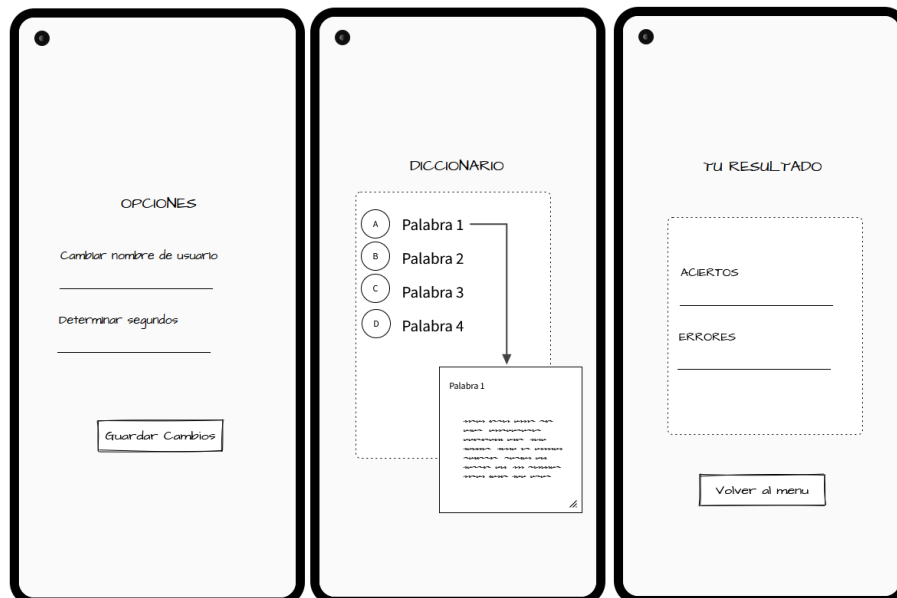
Para esto se realizó un primer mock up,





Como vemos en las imágenes, la aplicación iba a constar de una pantalla de Login, Menu, Juego, Añadir palabra, Opciones y Logout.

A través del desarrollo surgieron oportunidades de completar aún más la aplicación y se decidió añadir algunos cambios.

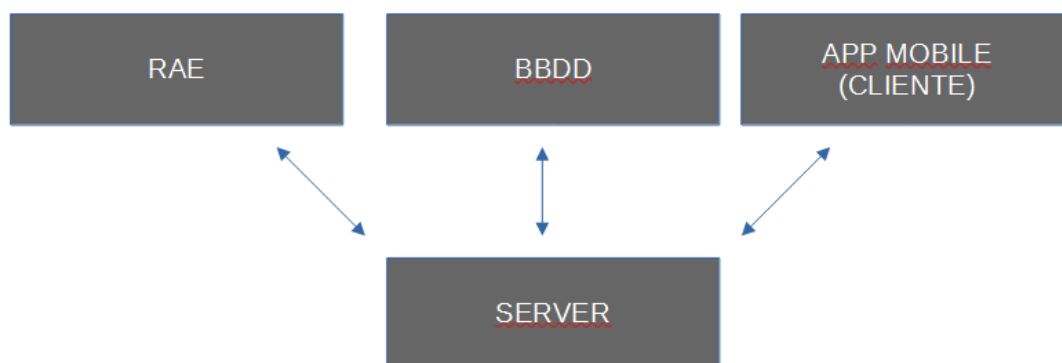


Se modificó la pantalla de opciones, se añadió una pantalla de diccionario y también una pantalla en la que el usuario pudiese ver su resultado.

De esta forma la aplicación no solo sería un juego, también podría hacer las veces de diccionario personal.

### 5.3 DESCRIPCIÓN Y ARQUITECTURA DEL PROYECTO.

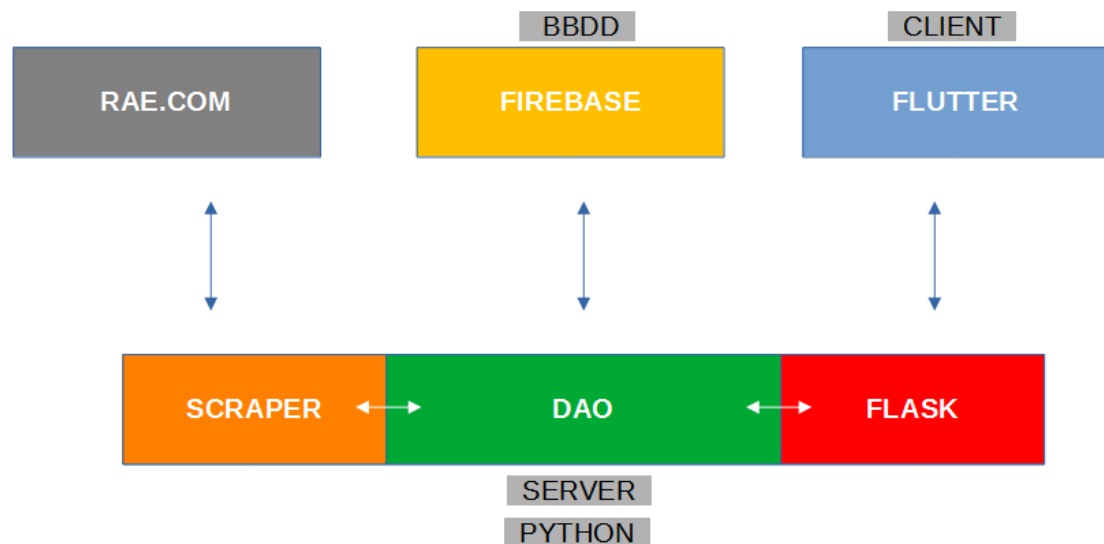
La arquitectura de este juego fue planteada con una arquitectura Cliente/Servidor, con este último vinculado a una base de datos NoSQL y que recolectara de manera automática las descripciones y las palabras.



La estructura del proyecto se dividió en tres componentes principales: la aplicación móvil desarrollada en Flutter, el servidor Flask en Python junto con el scraper dentro de este, y la base de datos Firebase.

Mediante el **scraper**, técnica que permite extraer datos de una página web de manera automatizada, la aplicación recopila información lingüística de la Real Academia Española (RAE) y la almacena en la BBDD. Para llevar a cabo esta tarea, se implementó un scraper alojado en el servidor Python que se encargó de realizar las solicitudes a la página web de la RAE y extraer la información deseada gracias a la librería **Beautiful Soup**.

La aplicación móvil actúa como cliente y es la encargada de enviar solicitudes al servidor Flask para obtener las palabras y definiciones requeridas al server (API REST). Estas solicitudes fueron gestionadas por el servidor Flask, que a su vez utilizó un objeto de acceso a datos (DAO, por sus siglas en inglés) para controlar los métodos relacionados con la obtención de datos de la página web, el almacenamiento de la información en Firebase y la gestión de los datos almacenados.



#### 5.4 DESARROLLO.

El desarrollo de la aplicación de esta manera ayudó a llevarlo a cabo de manera simultánea, lo que permitió un avance más eficiente y rápido. El frontend, es decir, la aplicación móvil, y el backend, representado por el servidor Python, fueron desarrollados en paralelo en las primeras instancias del proyecto. Para realizar pruebas y asegurar el correcto funcionamiento del servidor, se utilizaron las funciones que proporciona Flask para realizar las peticiones por medio de sencillas páginas en HTML de pruebas para ir mostrando la evolución mientras se desarrollaba la aplicación en Flutter. Además, se emplearon diversas librerías como Unittest, Locust y Request para la comprobación de pruebas y tenerlo listo hasta que la App en Flutter pudiera gestionar un uso básico de las funciones principales.

Este enfoque de desarrollo simultáneo permitió una mayor agilidad en el proceso, ya que se pudo iterar y realizar pruebas de manera constante, asegurando un correcto funcionamiento de la aplicación en todas sus partes. Además, el uso de librerías de pruebas facilitó la detección temprana de posibles mejoras en las funcionalidades y algunos errores o fallos en las respuestas.

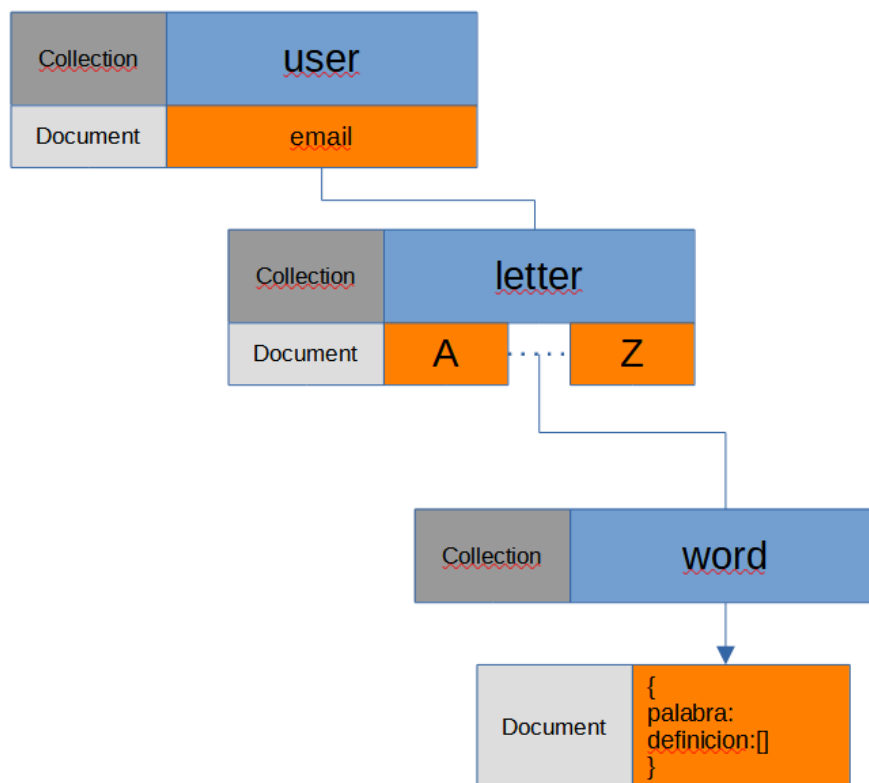
## 5.5 BBDD: FIREBASE.

La base de datos de FireBase es NoSQL y se estructura con una serie de colecciones y documentos dentro de estas.

Al tener cada usuario su diccionario propio se dictaminó que se almacenara de esta manera:

EMAIL → INICIAL → PALABRA

La decisión de diferenciar las palabras por inicial fue determinada por optimización. Al tener acceso directo a la inicial, se elimina buscar entre todas las palabras del diccionario y reduce ese cálculo en la iteración. Secundariamente, ya que el juego se basa en la búsqueda de las palabras, facilita el proceso de búsqueda al azar de las palabras dentro de cada letra.



### 5.5.1 Authentication.

Como se ha visto en FireBase y en el desarrollo de DAO, para la autenticación de usuarios se utilizó Authentication, una herramienta de Google que nos permite fácilmente la comprobación del login de usuario y el almacenamiento de las credenciales de manera codificada.

Para la configuración del FireBase authentication se importa desde dos JSON, uno para pyrabase:

```
# Configura la instancia de Firebase
import json

with open('firebase_config.json') as f:
    config = json.load(f)
```

Y otro para en el DAO para la configuración del Firebase\_admin:

```
#Conexión con firebase
cred = credentials.Certificate("serviceAccountKey.json")
default_app = firebase_admin.initialize_app(cred)
```

Aunque se pudo usar solo uno, por comodidad se utilizaron ambos (uno para log-in y otro para sign-in) gracias al uso de las diferentes librerías anteriormente mencionadas.

Estos documentos se pueden modificar en caso de que se quiera configurar una BBDD en FireBase alternativa.

## 5.6 SERVER.

El servidor, desarrollado en Python, se puede dividir en tres partes: Scraper (BeautifulSoup), DAO y API Rest (Flask).

Esta manera de trabajar nos permite localizar y corregir cualquier error de manera más precisa además de ayudarnos a poder realizar cambios severos de forma modulable. Por ejemplo, en el caso de cambiar de BBDD (de FireBase a una opción SQL) solo habría que modificar los métodos del DAO para su correcto funcionamiento (posiblemente la salida del Scraper pero también se podría mantener). O si en lugar de Flask se optase por otra opción como Django, solo habría que trabajar sobre esa parte para gestionar las peticiones del cliente de manera correcta.



### 5.6.1 SCRAPER

Este es el encargado de realizar las peticiones y organizar las respuestas para obtener los datos que se requieren para almacenarlos en la base de datos de FireBase. BeautifulSoup es un paquete que nos ayudará a parsear documentos HTML y XML. En este caso nos basaremos en el primero ya que la respuesta que nos dará [www.rae.es](http://www.rae.es) será de este tipo.

#### - ANALIZAR RAE.COM

Antes de realizar el scraper hay que analizar la web de la RAE para entender cuáles son las partes de información que vamos a necesitar y cómo organizarlas.



Este es un paso primordial para quedarnos con las clases o etiquetas del html que necesitamos para el almacenamiento.

Los datos principales que necesitamos para el almacenamiento sería palabra y listado de definiciones, por lo cual se optó en que este scraper devolviera un diccionario de datos ya que sería útil para el posterior almacenamiento en FireBase por ser una BBDD No SQL.

The image shows a screenshot of the Real Academia Española (RAE) dictionary website. The main content area displays the definition of 'ejemplo' (example), including its etymology and usage. A red arrow points from the definition text to the HTML code on the right, specifically highlighting the class 'j' and 'j1' which contain the definition text. The HTML code on the right shows the structure of the page, including the header, navigation menu, and the main content area. The class 'j' is used for the definition text, and 'j1' is used for the example text.

Las definiciones están alojadas en las clases j, j1 y j2 por lo cual era fácilmente identificables.

Para la búsqueda otra cosa que hubo que analizar fueron las direcciones de cada entrada del diccionario para desarrollar la llamada de forma correcta. Esta se compone de esta estructura:

**'https://dle.rae.es/' + word**

Esto nos permitió hacer las llamadas pertinentes según el parámetro de entrada a este Scraper.

Tras analizar los datos que necesitábamos de la página hubo que añadir a un header a las cabeceras para simular que se hacen desde un navegador y que la página admita la petición GET.

```
def myword(word):  
  
    url = 'https://dle.rae.es/' + word  
    print(url)  
  
    headers = {  
        'User-Agent': 'Mozilla/5.0(Windows NT 10.0; Win65; x64)AppleWebKit/537.36 (KHTML, like  
    response = requests.get(url, headers=headers)  
    soup = bs(response.content, 'html.parser')  
    definitions = soup.find_all('p', {'class': ['j', 'j1', 'j2']})  
  
    defs = []
```

Por último, hubo que reorganizar los datos, algo que Python junto con BeautifulSoup nos ayudará a quitar las etiquetas del HTML bruto y dejarlo en un diccionario de datos de este estilo:

```
defs_validas = []  
  
for i in defs:  
    defs_validas.append(i)  
  
def_string_val = '\n'.join(defs_validas)  
  
print(def_string_val)  
nueva_palabra = {  
    "palabra": word,  
    "definiciones": defs  
}  
return nueva_palabra
```

En Python, un archivo funciona básicamente como un objeto con funciones estáticas. Por lo que este Scraper es fácilmente invocable desde el DAO.

### 5.6.2 DAO

El DAO tiene la función de vincular las peticiones que nos llegan desde Flask para almacenarlas o extraer la información solicitada de la BBDD.

Esta clase reúne los métodos necesarios que gestionan esta vinculación. Alberga el scraper (que se llama como un objeto estático, incluso podría ser un método dentro de esta clase), y todos los métodos de login, registro, creación de usuario, almacenamiento de palabras, extracción de palabras, definiciones y extracción de palabras al azar que será la clave del juego.

La clave del desarrollo de este DAO es el entendimiento de la navegación de Firebase, explicado en el punto anterior. El acceso a la información que se debe extraer tiene que ser bien especificada en cada llamada:

```
def get_def(self, user_email, word):
    db = firestore.client()
    def_list = db.collection('users').document(user_email).collection('letter').document(unicode(word[0])).collection('word').document(word).get().to_dict().get('definiciones', [])
    print(def_list)
    return def_list
```

En este método se extrae una colección definición con unos parámetros de entrada de email y palabra que son enviados en formato JSON desde la App y recibidos por el Flask que ejecutará este método.

Los métodos de Registro y Login están creados con la ayuda de las librerías firebase\_admin y pyrebase.

Uno de los métodos principales de la aplicación es la creación del juego, en él cada jugador deberá recibir una palabra y una definición de esta al azar por lo que hay que controlar la navegación en las colecciones de Firebase de manera eficiente:

```

def get_random_words(self, user_email):
    db = firestore.client()
    word_list=[]
    for letter in letters:
        word_ref = db.collection('users').document(user_email).collection('letter').document(letter).collection('word').get()
        if word_ref:
            word_doc = random.choice(word_ref)
            word = word_doc.id
            definitions = word_doc.to_dict().get('definiciones', [])
            if definitions:
                random_definition = random.choice(definitions)
                word_dict = {"palabra": word, "definicion": random_definition}
                word_list.append(word_dict)
                print(word_list)
            else:
                print(f"No se encontraron definiciones para la palabra {word}.")
        else:
            print(f"No se encontraron palabras para la letra {letter}.")
    print(word_list)
    return word_list

```

Este método anida un bucle FOR en el que el que navegará por todas las letras del listado y se busca la palabra al azar, gracias al random de Python, y después dentro de cada palabra seleccionada, se buscará una definición de la misma manera. Esto devolverá un diccionario para facilitar la el envío de esta información en un JSON para que la App pueda utilizarla.

### 5.6.3 FLASK

Esta parte del código aloja la aplicación (web en unos inicios) desarrollada con el framework Flask. Esta maneja la autenticación de usuarios, el registro, inicio de sesión, y la gestión de un diccionario de palabras.

Esto constituye el API Rest para gestionar las llamadas de los clientes. Flask permite varias modalidades como el trabajo con hilos y nos facilita la gestión de login (este uso actualmente fue descartado, aunque permanecen funciones ya que está orientado a web principalmente). Por último, una de las principales características que nos otorga es el uso de decoradores para una gestión más sencilla de las peticiones.

Esto nos permite que recibamos la llamada y ejecutar el método correspondiente:

```

@app.route('/register', methods=['POST'])
def create():
    if 'email' not in request.json or 'password' not in request.json:
        return "Error: faltan datos requeridos (email y/o password)", 400
    email = request.json['email']
    password = request.json['password']
    print(request.json)

    if user_dao.create_user(email, password)[0]:
        print(f'El usuario {email} se ha creado.')
        for word in default_words:
            user_dao.add_word_to_dic(email, word)
    else:
        return jsonify(["user already exist"])
    if authenticate_and_login_user(email, password):
        return jsonify(["Register success"])
    else:
        return jsonify(["Wrong Credentials"])

```

Flask por defecto nos habilita un localhost en el puerto 5000, esto sería editable si se alojara en un servidor final. Los decoradores nos ayudan a configurar estos endpoints, por ejemplo, las llamadas para este método serían dirigidas a esta dirección URL:

<https://localhost:5000/register>

Y en el body recibiríamos los parámetros de la petición en JSON. Para trabajar con JSON se utilizó la librería **jsonify**, la cual nos ayudaría a traducir el contenido de la información de las peticiones y almacenarlas en las variables para trabajar con ellas en el servidor (llamadas a los métodos del DAO).

Además, también devolverá la información de los errores que puedan surgir y así informar al cliente.

```
@app.route('/add', methods=['POST'])
def add():
    try:
        user_email = request.json['email']
        word_f = request.json['word']
        if user_dao.check_word_dict(user_email, word_f):
            word = user_dao.add_word_to_dic(user_email, word_f)
            if word is not None: # Verificar si words es None
                return jsonify(word_f)
            else:
                return 'No hay palabra', 400
        else:
            return jsonify(["Palabra " + word_f + " ya existe"])
    except Exception as e:
        app.logger.error(str(e))
        return 'Error: ' + str(e), 500
```

Todas las peticiones de protocolo HTTP serán con método POST ya que se envían los datos como mínimo el email del usuario que es el identificador que utilizaremos para todas las gestiones.

### 5.7 CLIENTE.

Para el envío de las peticiones HTTP a Flask se creó una clase que alberga todas las peticiones.

La principal función de este paquete es ayudar al envío de las peticiones HTTP al servidor Flask, por lo que debemos redirigir todas estas peticiones a la dirección 10.0.2.2. Esto se debe a que el emulador de Flutter se ejecuta detrás de un servicio enrutador/firewall virtual que lo aísla de las interfaces y configuraciones de red de tu máquina de desarrollo, así como de Internet.

Un dispositivo emulado no puede ver la máquina de desarrollo ni otras instancias del emulador en la red. En cambio solo ve que está conectado a través de Ethernet a un enrutador/firewall.

El enrutador virtual para cada instancia todas las direcciones administradas por el enrutador tienen la forma 10.0.2.<XX>.

En resumen, es un alias del host al estar encapsulado por la máquina virtual.

A continuación, vamos a explicar cómo se creó la clase `HttpService()`.

Lo primero fue crear las variables para poder hacer las llamadas al servidor,

```
class HttpService {
    static final _client = http.Client();
    static final localhost = "10.0.2.2";
    static var _loginUrl = Uri.parse('http://' + localhost + ':5000/login');
    static var _registerUrl = Uri.parse('http://' + localhost + ':5000/register');
    static var _dictionaryUrl =
        Uri.parse('http://' + localhost + ':5000/dictionary');
    static var _gameUrl = Uri.parse('http://' + localhost + ':5000/game');
    static var _addUrl = Uri.parse('http://' + localhost + ':5000/add');
    static var _deleteUrl = Uri.parse('http://' + localhost + ':5000/delete');
    static var _defUrl = Uri.parse('http://' + localhost + ':5000/definitions');
    static var _logoutUrl = Uri.parse('http://' + localhost + ':5000/logout');
```

Aquí vemos la creación de la variable `_client`, se utiliza como cliente HTTP para enviar las solicitudes, la clase `Client` se encarga de mantener las conexiones persistentes a través de múltiples peticiones al mismo servidor.

También vemos la creación de las variables `_loginUrl`, `_registerUrl`, `_gameUrl`, entre otras variables que se usan para la llamar a los metodos creados en el servidor, como hemos explicado en el apartado de DAO.

Este código define variables estáticas que representan las URLs utilizadas para las solicitudes HTTP en diferentes endpoints del server. Esto permite un acceso fácil y una gestión centralizada de las URLs.

## 5.8 APP MOBILE: FLUTTER.

La aplicación móvil se desarrolló en Flutter gracias al uso del lenguaje DART.

Lo que hace diferente a Flutter es el uso de Widgets, estos son construidos usando un moderno framework inspirado en React, la idea principal es que se construya la interface con Widgets.

Estos son objetos temporales, se usan para construir una representación de la aplicación en su estado actual.

Existen dos tipos de Widgets,



Los widgets tienen una jerarquía y se componen de otros, cada uno se integra en el interior de otro.



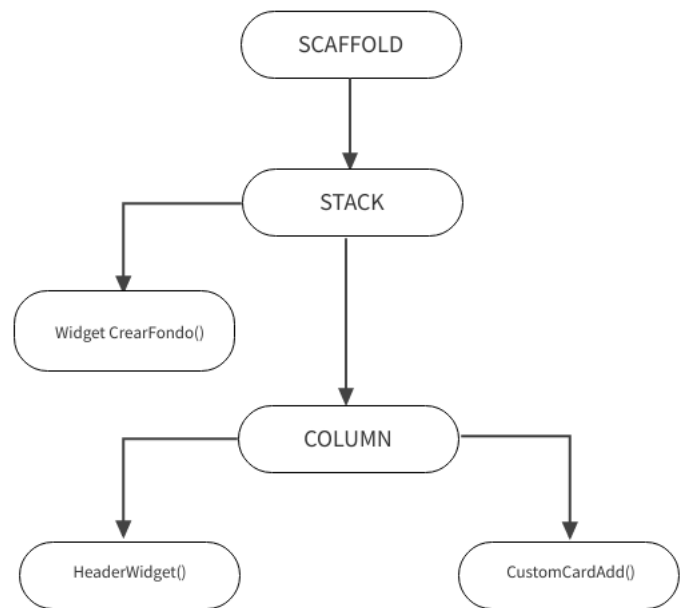


<b>Row / Column</b> Permiten crear layouts flexibles tanto horizontal (Row) como vertical (Column).	<b>Stack</b> Permite apilar los widgets uno encima de otro en el orden en el que se pintan.
<b>Container</b> Permite crear un elemento visual rectangular, puede ser decorado y contener márgenes.	<b>ListView</b> Es el widget más usado con scrolling, muestra sus hijos uno después de otro en dirección del scroll.
<b>Padding</b> Inserta sus hijos con el padding dado.	<b>Text</b> Muestra una cadena de texto con un estilo único.
<b>Center</b> Centra a sus hijos en si mismo.	<b>Swiper</b> Contiene un carrusel táctil para que el usuario pueda deslizar cualquier vista, en este caso las tarjetas. Usando esta librería podemos usar diferentes tipos de layouts.
<b>SingleChildScrollView</b> Caja en la que un widget puede ser scrollable.	<b>GestureDetector</b> Este detecta gestos y lleva a cabo las actividades descritas.
<b>StreamBuilder</b> Permite hacer un rebuild del componente cuando ocurre un nuevo evento. Por cada evento que recibamos devuelve un widget con la información actualizada.	<b>GestureDetector</b> Este detecta gestos y lleva a cabo las actividades descritas.

De esta manera, al crear widgets se pueden reutilizar a lo largo de la App para facilitar y optimizar el código.

### 5.8.1 Creación pantallas con Flutter.

Por ejemplo, el esquema de widgets de la siguiente pantalla es,



A continuación, detallaremos las partes principales del código de la pantalla anterior, como vemos en la imagen de abajo,

En este caso nos encontramos con un widget del tipo de **StatelessWidget**, esto es, que no se puede alterar una vez construido.

```

class AddPalabra extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Stack(children: [
        crearFondo(
          context,
        ),
        SingleChildScrollView(
          child: Column(children: const <Widget>[
            HeaderWidgetAdd(),
            SizedBox(height: 50.0),
            CustomCardAdd(),
          ]), // <Widget>[] // Column
        ) // SingleChildScrollView
      ]), // Stack
    ); // Scaffold
  }
}

```

El método build devuelve un **Scaffold** widget, a su vez tiene como hijo un **Stack** widget para poder colocar, por un lado, el fondo **crearFondo()** y el contenido principal.

El contenido principal está contenido dentro de un column, que organiza sus hijos en una columna vertical.

El widget column tiene tres hijos,

- 5 **HeaderWidgetAdd**, explicaremos su funcionamiento en puntos posteriores.
- 6 **SizedBox**, es un widget que añade espacio vertical entre el HeaderWidgetAdd y CustomCardAdd.
- 7 **CustomCardAdd**, explicaremos también mas adelante su funcionamiento.

En conclusión podemos ver que estos 4 widgets construyen la pantalla AddPalabra, que como vemos consta de un fondo personalizado, un encabezado y un widget personalizado.

## HeaderWidgetAdd

Creamos un Widget que hará las veces de AppBar, el Widget devuelve un Padding, que agrega un espacio relleno alrededor del hijo.

```
Widget build(BuildContext context) {  
  return Padding(  
    padding: const EdgeInsets.all(26.0),  
    child: Column(crossAxisAlignment: CrossAxisAlignment.start, children: [  
      Text(  
        "Añade tu palabra",  
        style: TextStyle(  
          color: titleTextColor,  
          fontSize: 40,  
          fontWeight: FontWeight.w900,  
          fontFamily: 'Avenir'), // TextStyle  
        textAlign: TextAlign.left,  
      ), // Text  
      Row(  
        crossAxisAlignment: CrossAxisAlignment.center,  
        children: [  
          IconButton(  
            icon: const Icon(Icons.arrow_back_ios_rounded),  
            onPressed: () => Navigator.pushReplacementNamed(context, 'home'),  
          ) // IconButton  
        ],  
      ), // Row  
    ]), // Column  
  ); // Padding  
}
```

El hijo del widget Padding es un widget Column, esto organiza sus hijos verticalmente. Usamos la propiedad crossAxisAlignment para alinear el contenido al comienzo de la columna, estableciendo esta característica con CrossAxisAlignment.start.

La columna tiene dos hijos,

1. Un widget Text que muestra el texto que queremos mostrar, en este caso 'Añade tu palabra'. Le damos estilo al widget con style, cambiamos el color, tamaño de fuente, entre otras características.
2. Un widget Row que organiza sus hijos en una fila horizontal. A su vez esta fila tiene un hijo, que es un widget IconButton, que muestra un icono de flecha hacia atrás.

Como vemos, el IconButton tiene dos propiedades; el icono que elegimos, Flutter tiene gran variedad de Iconos. Y, por otro lado, la propiedad onPressed, en la que se realiza una navegación a la página de Home a través de Navigator.pushReplacementNamed.

## CustomCardAdd

Esta clase extiende de StatefulWidget, esta clase representa un widget interactivo y mutable que muestra un card personalizado en la pantalla.

```
class CustomCardAdd extends StatefulWidget {  
  const CustomCardAdd({Key? key}) : super(key: key);
```

Se inicializa un controlador de texto, \_palabraController, que gestionará la entrada de texto del usuario y se usará el método dispose para liberar recursos, se llama cuando el widget se elimine.

Algo a tener en cuenta en el caso de \_palabraController, el guión bajo delante del nombre de la variable hace que esta sea privada. Se usa para variables y para funciones.

```
class _CustomCardAddState extends State<CustomCardAdd> {  
  final _palabraController = TextEditingController();  
  
  @override  
  void dispose() {  
    _palabraController.dispose();  
    super.dispose();  
  }
```

El contenido principal del widget se almacenará dentro de un container widget , que es flexible y configurable, que tiene un hijo de tipo card widget que mostrará una card en la pantalla, con bordes redondeados y sombreado suave.

Instanciamos el objeto bloc, que viene de una clase contenida en la aplicación llamada Provider.

```
Widget build(BuildContext context) {
  final bloc = Provider.of(context);
  return Container(
    child: Card(
      elevation: 8,
      shape: RoundedRectangleBorder(borderRadius: BorderRadius.circular(32)),
      color: Colors.white,
```

Uno de los elementos de la card será un TextField en el que se ingresará la palabra que queremos añadir al diccionario. El controlador de texto `_palabraController` captura los datos que el usuario introduce.

```
Padding(
  padding: const EdgeInsets.only(top: 10),
  child: TextField(
    controller: _palabraController,
    decoration: InputDecoration(
      hintText: 'Ingresa tu palabra',
      border: OutlineInputBorder(
        borderRadius: BorderRadius.circular(8)),
    ), // TextField
  ), // Padding
```

Otro elemento importante en la card será el botón que añadirá la palabra al diccionario. En este caso será `ElevatedButton` que se activa al presionarlo y ejecuta una acción,

```
ElevatedButton(
  onPressed: () async {
    final palabra = _palabraController.text;
    EasyLoading.show();
    await HttpService.add(bloc.email, palabra, context);
    EasyLoading.dismiss();
  },
```

esta acción será el método `add()` contenido en la clase `http.dart`,

```

static add(email, word, context) async {
    Map data = {'email': email, 'word': word};
    http.Response response = await _client.post(_addUrl,
        body: jsonEncode(data), headers: {'Content-Type': 'application/json'});
    if (response.statusCode == 200) {
        await EasyLoading.showSuccess("Palabra " + word + " añadida");
    } else {
        await EasyLoading.showError(
            "Error Code: ${response.statusCode.toString()}");
    }
}
}

```

En esta parte del código se puede ver una característica importante de la aplicación, el uso de funciones asíncronas con el uso de async. Como podemos ver también se usa await, con la finalidad de esperar a que se complete la tarea asíncrona y pueda continuar con su ejecución.

Este método estático, que recibe el email, la palabra y un contexto, realiza una solicitud HTTP para agregar la palabra al server, se realiza una solicitud POST usando la clase \_cliente que representa al cliente HTTP. La información se envía con formato JSON usado jsonEncode, esto convierte el mapa data en una cadena JSON.

## 5.9 SCREENS PRINCIPALES

### 5.9.1 Pantalla de Juego.

La screen principal sería la del juego. Esta consta principalmente de un Swiper en la que carga toda la información tanto de palabra y de definiciones. El widget de tarjeta se construye con esta información obteniendo así la inicial y la definición para el juego.

```
Expanded(  
  child: SingleChildScrollView(  
    child: Container(  
      constraints: BoxConstraints(  
        maxHeight: MediaQuery.of(context).size.height,  
        maxWidth: MediaQuery.of(context).size.width,  
      ), // BoxConstraints  
      child: Swiper(  
        controller: _swiperController,  
        itemCount: palabras.length,  
        itemWidth: MediaQuery.of(context).size.width,  
        itemHeight: MediaQuery.of(context).size.height,  
        layout: SwiperLayout.TINDER,  
        itemBuilder: (context, index) {  
          return IndexedStack(  
            children: [  
              Column(  
                children: [  
                  CustomCard(  
                    // añadir key para automatizar la actualización de Status  
                    key: Key(palabras[index]),  
                    palabra: palabras[index],  
                    definicion: definiciones[index],  
                    eliminarTarjeta: () => eliminarTarjeta(index),  
                    sumarAcierto: () => sumarAcierto(),  
                    sumarFallo: () => sumarFallo(),  
                  ), // CustomCard  
                ],  
              ),  
            ],  
          ),  
        },  
      ),  
    ),  
  ),  
),
```

Las cartas se añaden en el swiper según el array de entrada (las palabras que lo incorporan) y se crea, con un Layout predefinido (TINDER), una pila (IndexedStack) que organiza las tarjetas y además a cada una se le añade una serie de funciones para controlar los aciertos (variable global en la clase), eliminar la tarjeta tras acierto o fallo.



```

@override
void didChangeDependencies() {
  super.didChangeDependencies();
  List<List<String>> palabras_def =
    ModalRoute.of(context)?.settings.arguments as List<List<String>>;
  palabras = palabras_def[0];
  definiciones = palabras_def[1];
}

@override
void initState() {
  super.initState();
}

void eliminarTarjeta(int index) {
  setState(() {
    palabras.removeAt(index);
    definiciones.removeAt(index);
  });
  if (palabras.isEmpty) {
    Navigator.push(
      context,
      MaterialPageRoute(
        builder: (context) =>
          ScorePage(aciertos: contadorAcierto, fallos: contadorFallo)), // MaterialPageRoute
    );
  }
}

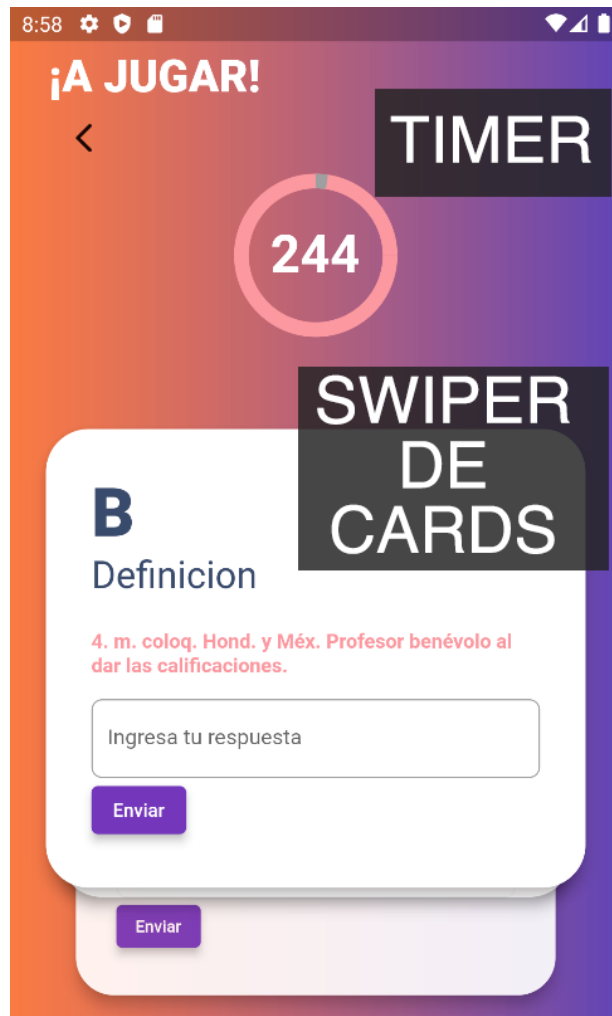
void sumarAcierto() {
  setState(() {
    print("Se ha acertado");
    contadorAcierto++;
  });
}

void sumarFallo() {
  setState(() {
    print("Se ha fallado");
    contadorFallo++;
  });
}

```

Los métodos principalmente gestionan el array de tarjeta y hacen llamadas para usar los métodos de animación que tiene cada widget tarjeta.

Los métodos son llamados en la misma clase, siendo eliminar tarjeta el más importante.



## STATEFUL WIDGET:

Como se explicó antes, los statefulwidget necesitan controlar los estados y con el método de la interfaz `didChangeDependencies` podemos controlar los estados y así actualizar las tarjetas eliminadas.

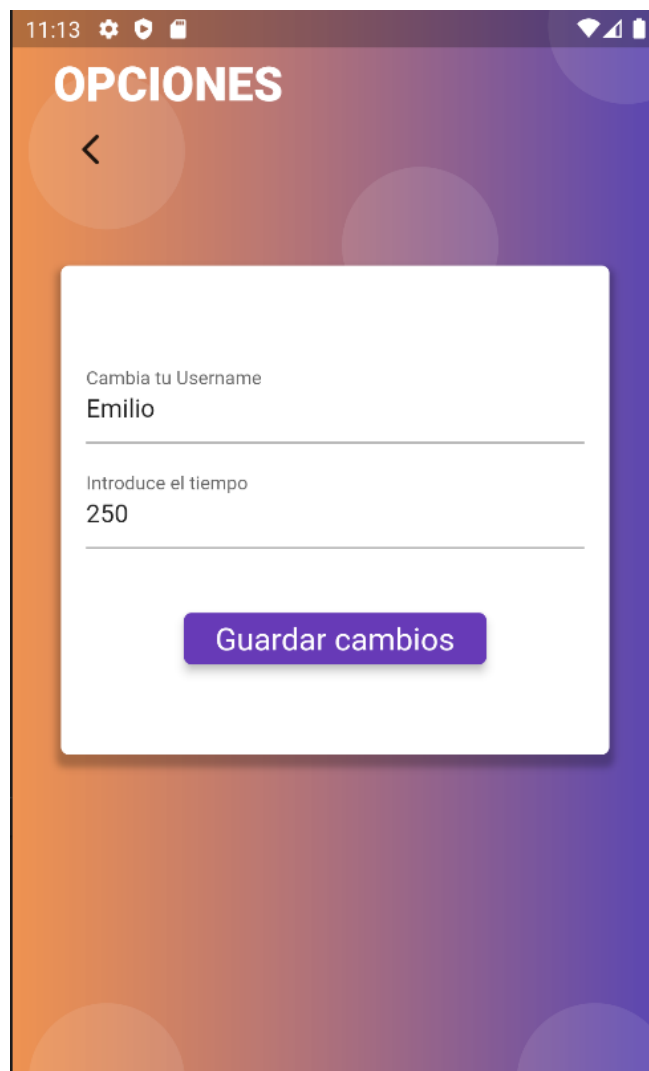
Finalmente, tendríamos también un widget con el Timer que tenemos en otra clase e importamos con éxito para utilizarla aquí y el juego acabará cuando el tiempo se acabe o se acaben las tarjetas pasando al screen de resultado.

```
onTimerFinish: () {
  Navigator.push(
    context,
    MaterialPageRoute(
      builder: (context) => ScorePage(
        aciertos: contadorAcierto, fallos: contadorFallo)), // ScorePage // MaterialPageRoute
  );
}
```

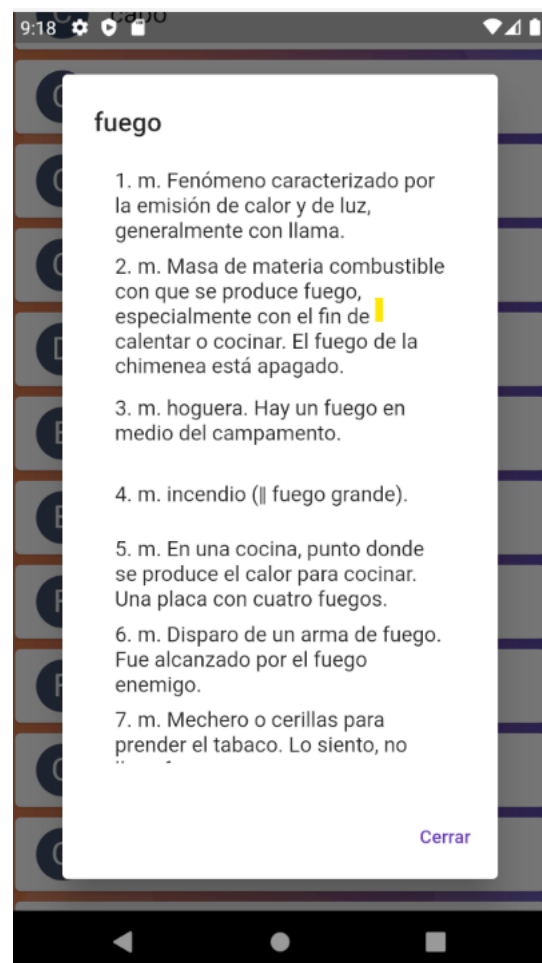
### 5.9.2 Pantalla Opciones.

En la anterior pantalla el timer recibe una entrada de tiempo que podemos variar en la página de opciones.

Esta variable se almacenará en el dispositivo gracias al paquete 'shared\_preferences/shared\_preferences.dart'. La información se guarda en DATA/data/{application package} y así lo podemos utilizar tanto en la home para usarlo como bienvenida para el usuario y los segundos para utilizarlos y customizar el juego.



### 5.9.3 Pantalla Diccionario.



En esta pantalla usamos el widget ListView, se usa para construir una lista dinámica de elementos en función de la longitud de la lista de palabras. Es eficiente en cuanto a rendimiento, ya que solo renderiza los elementos visibles en pantalla.

```

ListView.builder(
  itemCount: palabras.length,
  itemBuilder: (BuildContext context, int index) {
    String inicial =
      removeDiacritics(palabras[index][0].toUpperCase());
    return GestureDetector(
      onTap: () async {
        print(palabras[index]);
        var definiciones = await HttpService.get_def(
          bloc.email, palabras[index], context);
        // ignore: use_build_context_synchronously
        _mostrarVentanaEmergente(
          context, palabras[index], definiciones);
      },
      child: Dismissible(
        key: ValueKey(palabras[index].toString()),
        background: Container(
          color: Colors.redAccent,
          child: Icon(Icons.delete, color: Colors.white, size: 40),
          alignment: Alignment.centerRight,
          padding: const EdgeInsets.symmetric(horizontal: 40),
        ), // Container
        direction: DismissDirection.endToStart,
        onDismissed: (direction) {
          HttpService.delete(
            bloc.email, palabras[index].toString(), context);
        },
      ),
    );
  },
);

```

Este widget usa un itemCount que es el tamaño de la lista, la propiedad itemBuilder para definir cómo se construye cada elemento de la lista. Dentro del itemBuilder se obtiene la primera letra de cada palabra de la lista palabras usando removeDiacritics y se le asigna a la variable inicial.

Con GestureDetector, se detecta cuando se toca cada elemento de la lista. Al tocar cada elemento se llama a la función mostrarVentanaEmergente, que muestra la palabra y la lista de definiciones de la misma.

Cada elemento del ListView está envuelto en un widget Dismissible, este widget convierte su hijo en descartable cuando se desliza en la dirección indicada, en este caso a la izquierda. Cuando se desliza a la izquierda se realiza una solicitud HTTP para eliminar el elemento de la lista, método contenido en la clase HttpService().

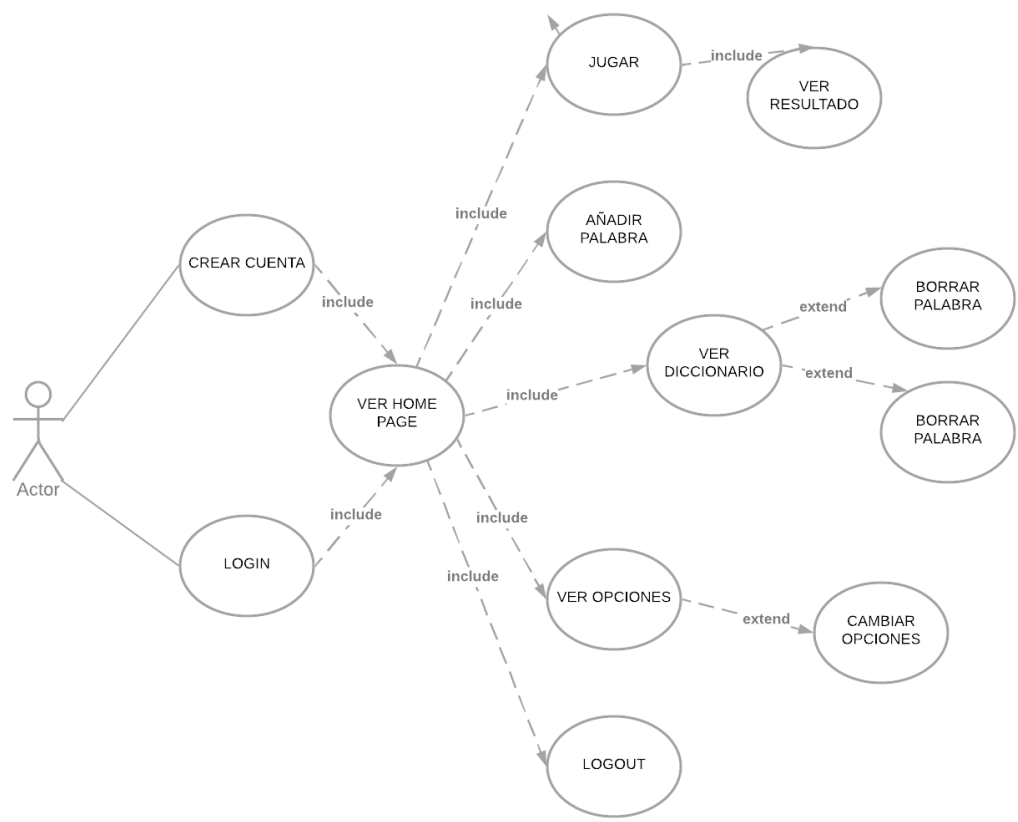
La ventana emergente que aparece al darle a cualquier elemento del diccionario se crea con el código que vemos a continuación,

```
void _mostrarVentanaEmergente(  
  BuildContext context, String word, List<String> definiciones) {  
  showDialog(  
    context: context,  
    builder: (BuildContext context) {  
      return AlertDialog(  
        title: Text(word),  
        content: SizedBox(  
          width: double.maxFinite,  
          child: ListView.builder(  
            //Ajustar automáticamente la altura del ListView  
            shrinkWrap: true,  
            itemCount: definiciones.length,  
            itemBuilder: (BuildContext context, int index) {  
              return ListTile(  
                title: Text(definiciones[index]),  
              ); // ListTile  
            },  
          ), // ListView.builder  
        ), // SizedBox  
        actions: <Widget>[  
          TextButton(  
            child: Text('Cerrar'),  
            onPressed: () {  
              Navigator.of(context).pop();  
            },  
          ), // TextButton  
        ], // <Widget>[]  
      ); // AlertDialog  
    },  
  );  
}
```

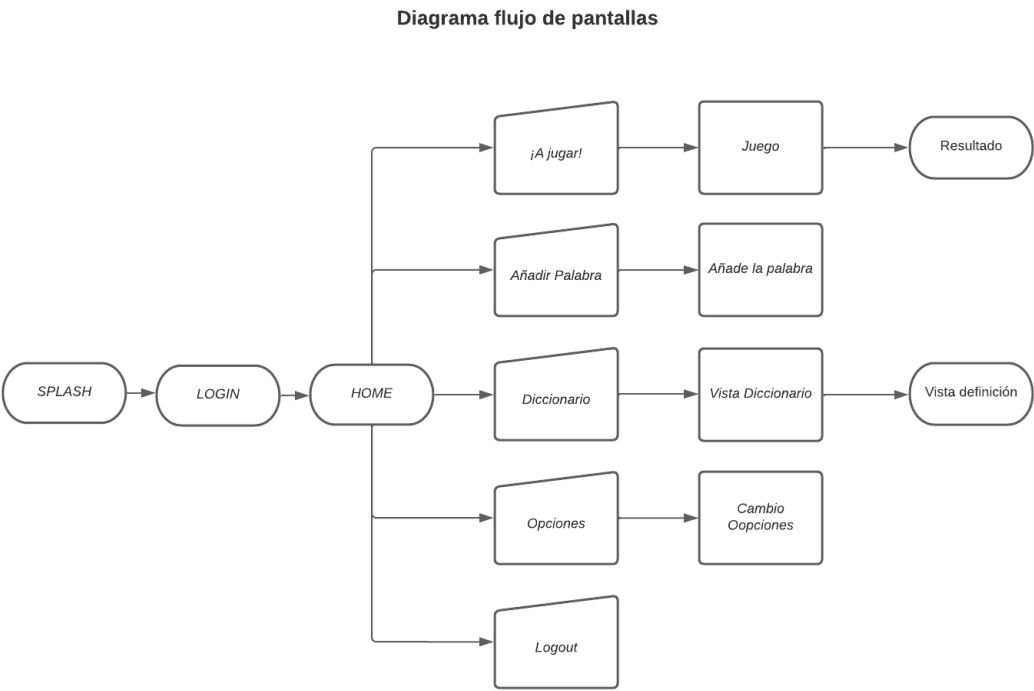
Se utiliza showDialog para mostrar un AlertDialog, este último se retorna del builder y tiene el title, que en este caso es la palabra que se ha presionado y el content que es un ListView que contiene las definiciones de la palabra.

Se muestra también un botón para cerrar la ventana emergente.

5.10 DIAGRAMA DE CASOS DE USO.



5.11 DIAGRAMA DE PANTALLAS.





## 6. Conclusiones y mejoras.

---

### 6.1 Conclusión.

Tras desarrollar esta aplicación hemos llegado a la conclusión de que Flutter permite crear aplicaciones modernas y atractivas con cierta facilidad. Algo muy importante es la existencia de documentación y de una comunidad muy activa en la que apoyarnos para solucionar y desarrollar todo lo que podamos imaginarnos.

La combinación de Flutter, Flask y Firebase nos ha dado una solución completa y eficiente para poder desarrollar la aplicación. Estas tecnologías nos han proporcionado herramientas poderosas que hacen más fácil la creación de una interfaz atractiva, un backend escalable y una gestión simple de los datos.

Aunque, el haber desarrollado la aplicación con un servidor implica mayor complejidad de desarrollo y dependencia de la conexión a Internet (en el caso de producto final), este método ofrece ventajas como la escalabilidad, pudiendo realizar optimizaciones en la parte lógica sin tener que requerir acciones por parte del usuario con actualizaciones y mantenimiento centralizados en el servidor. Además ofrece la posibilidad de poder interconectar varios usuarios entre sí, una mejor gestión de datos y seguridad más sólida. Asimismo, permite un ahorro de recursos en los dispositivos móviles al trasladar parte de la lógica al servidor.

### 6.3 Mejoras.

La aplicación puede mejorarse de varias maneras, y las principales que se han detectado ya que se priorizó un funcionamiento completo a estos detalles aunque importantes fueron las siguientes:

- Quitar definiciones que contengan la misma palabra (facilitaría el juego)
- Palabras con varios géneros (detectar la definición corresponde al término adecuado)
- Seguridad: por ahora no se incluyó un sistema de logueo en el servidor. En un principio se configuró para la versión API web en las primeras prueba, pero con

Flutter descubrimos incompatibilidades y no se integró un sistema de tokenización para el acceso o de cookies.

## 6.2 Escalabilidad.

La aplicación tiene posibles mejoras que se podrían hacer en el tiempo y adquiriendo un mayor conocimiento. Estas podrían ser:

- Integración con redes sociales. Se podría agregar la posibilidad de compartir el progreso en el juego en redes sociales como Facebook o Twitter, permitiendo que los usuarios puedan compartir su experiencia de juego y desafiar a sus amigos a jugar también.
- Añadir otros idiomas. Expandir la aplicación para añadir palabras y definiciones en otros idiomas.
- Integración con asistente virtual. Integrar la aplicación con asistentes virtuales como Siri o Google Assistant, lo que permitiría que los usuarios jueguen usando comandos de voz.
- Jugar en línea. Agregar un modo en línea donde los usuarios puedan competir en tiempo real con otros jugadores, esto permitiría una experiencia de juego más interactiva y social.
- Incorporación de inteligencia artificial. Se podría usar para mejorar la experiencia de juego, sugiriendo palabras más difíciles o adaptando el nivel de dificultad de acuerdo al desempeño del usuario.
- Modos de juego temáticos. Agregar modos de juego temáticos para adaptarse a diferentes intereses y preferencias de los usuarios, por ejemplo, juego de palabras relacionadas con deportes, tecnología, entre otros.

## 7. Bibliografía.

---

Flutter

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

Flutter repositorio de paquetes

<https://pub.dev/>

Dart

<https://dart.dev/guides>

Mockup

<https://wireframepro.mockflow.com/>

Canva

<https://www.canva.com/>

Diagrama

<https://lucid.app/>

Diccionario

<https://dle.rae.es/>

API TESTING

<https://www.postman.com/>

Firebase

<https://firebase.google.com/>

Python

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

StackOverFlow

<https://stackoverflow.com/questions>

Conexión Flask – Flutter

<https://blog.logrocket.com/integrating-flask-flutter-apps/>

<https://www.section.io/engineering-education/flutter-authentication-using-flask-api/>

Flutter Cards

<https://medium.com/@prasathravi/universe-app-in-flutter-with-a-card-swiper-370109432e9a>

Shared Preferences

[https://pub.dev/packages/shared\\_preferences](https://pub.dev/packages/shared_preferences)

<https://medium.flutterdevs.com/using-sharedpreferences-in-flutter-251755f07127>

ListView swipe to dismiss

<https://docs.flutter.dev/cookbook/gestures/dismissible>

BeatyfulSoup

<https://realpython.com/beautiful-soup-web-scraper-python/>

<https://www.crummy.com/software/BeautifulSoup/bs4/doc/>

IP Emulador connection localhost

<https://stackoverflow.com/questions/5806220/how-to-connect-to-my-http-localhost-web-server-from-android-emulator>

