

**UNIVERSIDAD TÉCNICA PARTICULAR DE LOJA**

*La Universidad Católica de Loja*

**Carátula**

**FACULTAD DE INGENIERÍA**

**CARRERA DE INGENIERIA EN SISTEMAS INFORMÁTICOS Y COMPUTACIÓN**

**Manual del Programador   
Desarrollo e implementación de una plataforma tecnológica para el aprendizaje de patrones funcionales, necesidades básicas y dominios en estudiantes de enfermería**

Trabajo de titulación previo a la obtención del título de:

**INGENIERO EN SISTEMAS INFORMÁTICOS Y COMPUTACIÓN**

**Autor:** Celi Salinas, Marjan Aldair

**Director:** Irene Robalino, Pedro Daniel PhD

LOJA

2024

**Índice**

[Manual del Programador 4](#_Toc172493934)

[Introducción 4](#_Toc172493935)

[Objetivo 4](#_Toc172493936)

[Descripción General del Proyecto 4](#_Toc172493937)

[Tecnologías Utilizadas 4](#_Toc172493938)

[Desarrollo 4](#_Toc172493939)

[1. Sprint 1: Desarrollo del Backend 4](#_Toc172493940)

[1.1 Importación de Librerías 5](#_Toc172493941)

[1.2 Conexión a Firebase 5](#_Toc172493942)

[1.3 Subida de Información en formato JSON a Firebase 6](#_Toc172493943)

[1.4 Obtener elementos de necesidades 7](#_Toc172493944)

[1.5 Obtener elementos de patrones 7](#_Toc172493945)

[1.6 Obtener elementos de Dominios 8](#_Toc172493946)

[2. Sprint 2: Plataforma Web 8](#_Toc172493947)

[2.1 Configuración Inicial y Carga del DOM 8](#_Toc172493948)

[2.2 Transformar JSON 9](#_Toc172493949)

[2.3 Obtención y manejo de Información 10](#_Toc172493950)

[2.4 Login de Administrator 12](#_Toc172493951)

[3. Sprint 3: Chat Bot 12](#_Toc172493952)

[3.1 Configuración de la IA de Google y Flask 12](#_Toc172493953)

[3.2 Interacción del Chat Bot 13](#_Toc172493954)

[3.3 Almacenamiento de historial de Chat 14](#_Toc172493955)

[4. Sprint 4: Aplicativo Móvil 16](#_Toc172493956)

[4.1 Navegación 16](#_Toc172493957)

[4.2 Registro 17](#_Toc172493958)

[4.3 Autentificación 18](#_Toc172493959)

[4.4 Muestra de Información 19](#_Toc172493960)

[5. Requisitos del sistema 20](#_Toc172493961)

[Requisitos para la Aplicación Móvil 20](#_Toc172493962)

[Requisitos para la Aplicación Web 21](#_Toc172493963)

# **Introducción**

El propósito de este manual del programador es proporcionar una guía completa y detallada sobre el desarrollo, implementación y mantenimiento de la aplicación Florens. Florens es una aplicación diseñada específicamente para estudiantes de enfermería, ofreciendo herramientas educativas y de apoyo para facilitar su aprendizaje y mejorar su experiencia académica. Esta aplicación cuenta con un tutor automatizado, funcionalidades de chat con inteligencia artificial, y está respaldada por una base de datos no relacional en Firebase.

Este manual cubre aspectos clave del desarrollo, incluyendo la arquitectura del sistema, configuración del entorno de desarrollo, implementación del código, pruebas y despliegue. Su finalidad es asegurar que cualquier programador que se una al proyecto pueda entender y contribuir eficazmente al desarrollo y mantenimiento de la plataforma.

La plataforma desarrollada se basa en tecnologías web modernas y móviles, utilizando HTML, JavaScript y CSS para la estructura y estilo de la interfaz web, mientras que la aplicación móvil está construida con React Native. HTML proporciona la base estructural de las páginas web, JavaScript permite la implementación de la lógica y funcionalidad dinámica, y CSS se encarga del diseño visual y presentación.

Para la aplicación móvil, React Native facilita el desarrollo de aplicaciones nativas multiplataforma, aprovechando la potencia de React para crear interfaces de usuario eficientes y responsivas tanto para iOS como para Android. Esta combinación de tecnologías garantiza una experiencia de usuario cohesiva y de alto rendimiento en diferentes dispositivos y plataformas.

# **Arquitectura del Sistema**

Diagrama

Descripción generada automáticamente

**Descripción General**

El diagrama que me has proporcionado muestra la arquitectura de una aplicación web típica. La aplicación está dividida en tres capas principales:

**Capa de Presentación (FrontEnd)**

* **Tecnologías:** React Native, Web (HTML, CSS, JavaScript)
* **Componentes:**
  + **Aplicación:** La aplicación en sí, que es la interfaz de usuario con la que interactúan los usuarios.
  + **Administrador:** Un panel de administración para que los administradores puedan gestionar la aplicación.

**Capa Intermedia (Middleware)**

* **Tecnologías:** Flask (Python)
* **Componentes:**
  + **Capa de Acceso a Datos:** Proporciona acceso a la base de datos.
  + **Capa Lógica de Negocio:** Implementa la lógica de negocio de la aplicación.

**Capa de Datos (Base de Datos)**

* **Tecnologías:** Firebase
* **Base de datos:** Almacena los datos de la aplicación.

**Flujo de Datos**

El flujo de datos es el siguiente:

1. El usuario interactúa con la aplicación a través de la interfaz de usuario.
2. La aplicación envía una solicitud a la capa intermedia.
3. La capa intermedia accede a la base de datos a través de la capa de acceso a datos.
4. La capa lógica de negocio procesa los datos de la base de datos y genera una respuesta.
5. La capa intermedia envía la respuesta a la aplicación.
6. La aplicación muestra la respuesta al usuario.

# **Base de Datos**

Carta

Descripción generada automáticamente con confianza media

Esta es la estructura de base de datos Firebase para la plataforma Florens La base de datos está estructurada en dos niveles:

**Nivel 1: Colecciones**

* **Usuarios:** Esta colección almacena información sobre los usuarios de la aplicación.
* **Necesidades:** Esta colección almacena información sobre las necesidades. Cada documento en esta colección representa una necesidad.
* **Patrones:** Esta colección almacena información sobre los patrones. Cada documento en esta colección representa un patrón.
* **Dominios:** Esta colección almacena información sobre los dominios. Cada documento en esta colección representa un dominio.
* **Bibliografía:** Esta colección almacena información sobre la bibliografía utilizada en la aplicación. Cada documento en esta colección representa una referencia bibliográfica específica.

**Nivel 2: Documentos**

Cada documento en una colección tiene una estructura específica que define los campos que contiene. Por ejemplo, el documento de usuario puede contener los siguientes campos:

* **ID:** Un identificador único para el usuario.
* **Nombre:** El nombre del usuario.
* **Apellido:** El apellido del usuario.
* **Correo electrónico:** La dirección de correo electrónico del usuario.
* **Contraseña:** La contraseña del usuario.
* **Universidad:** La universidad a la que asiste el usuario.

**Relaciones entre documentos:**

Las relaciones entre documentos se pueden establecer utilizando campos de referencia. Por ejemplo, el documento de necesidad puede contener un campo de referencia al documento de usuario correspondiente. Esto significa que el documento de necesidad puede almacenar el ID del documento de usuario al que está asociado.

**Operaciones comunes:**

Las operaciones comunes que se pueden realizar en la base de datos incluyen:

* **Crear:** Crear un nuevo documento en una colección.
* **Leer:** Leer un documento existente en una colección.
* **Actualizar:** Actualizar un documento existente en una colección.
* **Eliminar:** Eliminar un documento existente en una colección.

Las cuales se encuentran dentro del portal web administrativo de manejo de información de la plataforma Florens.

# **4. Instalación y Configuración**

### **4.1 Requisitos del Sistema**

Hardware y Software necesarios para la instalar y ejecutar el sistema.

**Requisitos para la Aplicación Móvil**

* **Sistema Operativo:**
* Android: Versión 7.0 (Nougat) o superior
* iOS: Versión 11.0 o superior
* **Hardware:**
* Espacio de almacenamiento: Al menos 100 MB de espacio libre
* Memoria RAM: Mínimo 2 GB de RAM
* **Conectividad:**
* Conexión a Internet estable (Wi-Fi o datos móviles)
* **Permisos:**
* Acceso a Internet
* Permiso para recibir notificaciones push (opcional)
* **Herramientas de Desarrollo (para desarrolladores):**
* React Native: Framework principal para el desarrollo de la aplicación
* Expo CLI: Para la gestión y desarrollo de la aplicación
* Node.js: Versión 14 o superior
* Python: Para el backend y la integración con Firebase
* Firebase: Configuración de base de datos y autenticación

**Requisitos para la Aplicación Web**

Para poder trabajar en el desarrollo de la plataforma Florens, se requiere un entorno con las siguientes especificaciones mínimas:

* Sistema Operativo: Windows 10, macOS, o Linux
* Memoria RAM: 8 GB o más
* Espacio en Disco: 20 GB de espacio libre
* Software Necesario:
* Node.js (para el desarrollo móvil con React Native)
* Python 3.x (para el desarrollo del backend con Flask)
* Firebase CLI
* IDE o Editor de Código (Visual Studio Code, PyCharm, etc.)

### **Instalación y Configuración**

### **4.2.1 Instalación y Configuración del Entorno para la App**

* **Descargar e instalar la versión recomendada de Node.js, lo podemos encontrar en el siguiente enlace:** <https://nodejs.org/en/download/package-manager>**.**
* **Abrir la terminal y ejecutar “npm install -g expo-cli”.**
* **Crear un nuevo proyecto de React Native con expo ejecutando “expo init nombre\_proyecto”.**
* **Configurar Firebase: En la consola firebase se crea un nuevo proyecto y se agrega la aplicación.**
* **Descargar los archivos de configuración “google-services.json” y agregarlo a la carpeta de nuestro proyecto en la ruta (android/app).**
* **Instalamos la dependencia de Firebase “npm install firebase”.**
* **Instalamos todas las dependencias con el comando “npm install”.**

### **4.2.2 Instalación y Configuración de Entorno para la Web**

* Abrimos nuestro Entorno IDE para la realización del proyecto web.
* Instalamos el Firebase CLI ejecutando “npm install -g firebase-tools”.
* Iniciamos la sesión de Firebase con “firebase login”.
* Configuramos el proyecto en Firebase con “firebase init” para poder configurar el Firebase Hosting y Firestore.

### **4.2.3 Instalación y Configuración el Backend**

* Debemos tener instalo una versión de Python 3 o superior, en caso de no tenerla iremos a la pagina de descargar oficial en el siguiente enlace: [python.org](https://www.python.org/).
* Creamos el entorno virtual, navegamos en la carpeta de nuestro proyecto (Python -m venv venv).
* Activamos el entorno virtual “venv\Scripts\activate”.
* Instalamos Flask y otras Librerias, con el entorno virtual activado y ejecutamos “pip install flask flask-cors firebase-admin google-generativeai”.
* Configuramos Firebase, en la consola Firebase genera una clave de servicio (archivo.json), usamos este archivo para inicializar el proyecto Firebase Admin SDK:

import firebase\_admin

from firebase\_admin import credentials

cred = credentials.Certificate('path/to/serviceAccountKey.json')

firebase\_admin.initialize\_app(cred)

Como recomendación, asegúrese de verificar que la aplicación y portal web, estén correctamente conectados a Firestore mediante las credenciales proporcionadas.

### **4.2.4 Configuración de Key para Conexión con Firebase**

Para la comunicación del Backend de la Plataforma con la base de datos Firebase, y mas tema de seguridad no se subió la llave al repositorio para poder establecer la comunicación. A continuación, se detallan los pasos para configurar la llave y poder desplegar:

* Primero clonar proyecto de Florens
* Dirigirse a la carpeta FlorensBack y crear una carpita con el nombre “api”
* En la carpeta “api” crear un archivo con el nombre “key.json”.
* Agregaremos el siguiente código en el archivo y guardaremos y así podremos desplegar el Backend en nuestro entorno local.

{

  "type": "service\_account",

  "project\_id": "appenfermeria-d612d",

  "private\_key\_id": "85c6e5d0e9162475fbb1adfdd3e5f70cecb15f46",

  "private\_key": "-----BEGIN PRIVATE KEY-----\nMIIEvAIBADANBgkqhkiG9w0BAQEFAASCBKYwggSiAgEAAoIBAQCQjlWK7ORV4Ua2\n0pUVND27Wq8l5NfdpA7NeXljZNkgbUB0w/zPPckFOarES96+cZh51NVwpoSvjiKi\nDRLp7St9psiwZaDRxlLI+KVRHOcftx61BGpv7fjZ61BAz1GmMyhB9YLp5yJvD+u3\nEcT5CG9Nwnk+e0t0Ifz7E0P3wuLK4gBcLKP32fEJFyoVMzsep43U88R8Df/zb4oY\nOKjs9uf+9LUK4mC6gCF3ukKgxiwTGNg4T2Rk3Shj0PItnbo05evXJaH8AXZrzb/+\n9Sqzn2tTrmpfz2NZRsvwiXccsFRhSjJ40szjDlrmAlbfW6IC136PvRB40YfiK0M1\nnMxrRzsTAgMBAAECggEABsGQP/D4bwcbXfoTm60YYYRIgVXjvOkpA/N0d+/Dsh9J\nDuwGCc2ScpsSXyBIpc/7V15aJ2yw8N+MwNQpmSNLkBDWemAVlxhU0C3G1Dasmq/d\nUIqHSvntUooL8yt9Xm4vqH2IA9SYqGYvFxxOf3EkepbnQcNTcHcSMs3gtFi4btD1\nQLpi5txbIpYX/WLmKM+UecTT+7OJrXEdttM2LbGexT9st3WCRpZ+PXx6cL9v0Cwh\nxLpT07SbEabwVey+E/MHAAvvq8lF0lzX/Nwkng9lCKbnQHsQ/4kK905Mafg0oNMu\nkQKM4x0GTWZq/EUge3beYSzntADYA1vPb8HZP16BiQKBgQDJyh79rYuTOBF8vnCF\nyxGNBcKWmZeVEhUxO1b3pNI/vr95p8b2ULLi5CQwe6G9BhucoyoVTtjqV4BvxnkD\nVbYXI+E9Rz4Ol06QfVkrIwcA0JvvV+sFk2N56vqQbkBdgg3BotHgBoRGhvGRI3zl\nNVZ5mNmUtGIjB55edz6fyECYtQKBgQC3ZAgqEom1E7LNj1owoOrlCEApE5Agv4Vg\neS3U+kPgbSgc+XIGXLMD6AYSYH47xDJ6HF4DHFUU0o+yDcqzTcCCSTY1bI5zJMKF\nMt4m/ysWgxPTbKknJWoXEefaCedY/YHcjV1JqVA1bs01gQWJ2q4Uwcrr79me1w4J\n+NlI3jhJpwKBgDzOuLoPIwFRIwvpQ3U99WL+ul2xmRkl4tb8H7/uniDxdTfO1EkD\nlnSC9Vja2Klifs2b6aGUKg+sIJsOPwc/O+kLOePCZTP2aolmDInQO1sGggcFAmCi\nwpZJtgnx/1luSIDydLZCoHe0UHVBMXDMuutXoTGLjgsqT5H98xQpPzYFAoGAOhyh\nlatgJZgDXa3iN2yWYlL1+c4wKqBlxEdB5a4UGiacwmms3yU6uK8qVWJsPcYS9KzO\ngY/Vyg5a6GIEbELkEKTgX5dxoDzywsB2ssbCwZRxkNpq03PApWTQIkBnJcTXBKtq\n9y+GshfVQPIMbj10UjGCerb7L6EX2sMXH0cDcRsCgYAzfj7wadec2Nj5fZM1NMII\ntqKbnnFCjouKyOoPWLSsiX1g8ibKdyPph59OYmvLQNMtrNdX6/3ScLOkaCHWyYeH\njKd6NRAKMbJmzQmcpPM7fjKnmVxDJb4Bgh+/X3to38wSuhvesxS7t02R53fVk0RH\nX9HXhcWn3/16n2YuZTpZkA==\n-----END PRIVATE KEY-----\n",

  "client\_email": "firebase-adminsdk-75eow@appenfermeria-d612d.iam.gserviceaccount.com",

  "client\_id": "111094269452560552074",

  "auth\_uri": "https://accounts.google.com/o/oauth2/auth",

  "token\_uri": "https://oauth2.googleapis.com/token",

  "auth\_provider\_x509\_cert\_url": "https://www.googleapis.com/oauth2/v1/certs",

  "client\_x509\_cert\_url": "https://www.googleapis.com/robot/v1/metadata/x509/firebase-adminsdk-75eow%40appenfermeria-d612d.iam.gserviceaccount.com",

  "universe\_domain": "googleapis.com"

}

**Tecnologías Utilizadas**

* **Frontend Web**: HTML, CSS, JavaScript
* **Diseño**: Figma
* **Backend**: Python con Flask
* **Base de Datos:** Firebase Firestore
* **Autenticación**: Firebase Authentication
* **Aplicación** **Móvil**: React Native
* **Chat** **Bot**: API de Gemini de Google

# **5. Desarrollo**

## **5.1 Modulo 1: Desarrollo del Backend**

La plataforma está construida utilizando tecnologías web como HTML, JavaScript y CSS, mientras que la aplicación móvil está desarrollada con React Native. El backend de la plataforma está construido con Flask, una micro-framework para Python, y se conecta con Firebase para la gestión de datos y autenticación, además de utilizar la inteligencia artificial de Google para ciertas funcionalidades.

### **4.1.1 Importación de Librerías**

#Importacion de libreria para la creación de rutas  
from flask import Flask, jsonify, request  
#importacion de libreria para el uso de cors en el sistema  
from flask\_cors import CORS  
#Importacion de librerias para el uso de firebase  
import firebase\_admin  
from firebase\_admin import credentials, firestore, auth  
from firebase\_admin import exceptions  
#Importacion de la libreria para el uso de la IA de Google  
import google.generativeai as genai

*Nota: Para acceder al código completo, ir al GitHub:* [*https://lc.cx/vCWWxY*](https://lc.cx/vCWWxY)

Estas importaciones configuran el entorno de desarrollo del backend. Flask se usa para crear la aplicación web y definir rutas. Flask-CORS permite el acceso a la API desde distintos orígenes.

Firebase Admin SDK se usa para interactuar con los servicios de Firebase como Firestore y Authentication. Finalmente, la biblioteca de Google GenAI se usa para integrar la inteligencia artificial de Google.

### **5.1.2 Conexión a Firebase**

cred = credentials.Certificate('api/key.json')

firebase\_admin.initialize\_app(cred)

db = firestore.client()

*Nota: Para acceder al código completo, ir al GitHub:* [*https://lc.cx/vCWWxY*](https://lc.cx/vCWWxY)

Establece la conexión con Firebase utilizando las credenciales almacenadas en key.json. Se inicializa la aplicación Firebase y se obtiene un cliente Firestore para interactuar con la base de datos.

### **5.1.3 Subida de Información en Formato JSON a Firebase**

El programa en Python desarrollado tiene como objetivo agilizar y reducir el tiempo de carga de datos en la base de datos Firebase. Mediante técnicas de automatización y optimización del rendimiento, se busca mejorar la eficiencia en la gestión de datos, ofreciendo una solución rápida y efectiva para la carga de información en formato JSON

cred = credentials.Certificate('api/key.json')

firebase\_admin.initialize\_app(cred)

db = firestore.client()

data = {

 "Ingrese la información"

}

doc\_ref = db.collection("").document("").collection("").document("")

doc\_ref.set(data)

*Nota: Para acceder al código completo, ir al GitHub:* [*https://lc.cx/uvS-bo*](https://lc.cx/uvS-bo)

Para mejorar la rapidez de subida de información, en un solo tipo de formato y estructura, se optó por la realización de un programa para la optimización de esta, este código hecho en Python utiliza el Framework Flask para crear una API que interactúa con una base de datos Firestore de Firebase.

* Se crea una instancia de la aplicación Flask y se habilita CORS para permitir solicitudes de dominios cruzados.
* Se importan las librerías necesarias, como Flask para el desarrollo de la API, Flask-CORS para permitir solicitudes de recursos cruzados, y las bibliotecas de Firebase para interactuar con Firestore.
* Se establece una referencia al documento en la colección "Patrones" de Firestore, y se insertan los datos definidos en el paso anterior en un documento específico con la clave "Descripción".
* Se define un diccionario data que contiene la información a ser almacenada en la base de datos.

### **5.1.4 Obtener Elementos de Necesidades**

    items = db.collection('Necesidades').stream()

    print(items)

    datos\_coleccion = {}

    for i, doc in enumerate(items):

            # Usar el índice como ID comenzando desde 1

            identificador = str(i + 1)

            datos = doc.to\_dict()

            datos["Id"] = doc.id

            datos\_coleccion[identificador] = datos

    return jsonify(datos\_coleccion)

*Nota: Para acceder al código completo, ir al GitHub:* [*https://lc.cx/vCWWxY*](https://lc.cx/vCWWxY)

Este endpoint maneja las solicitudes GET a la ruta '/Necesidades'. Accede a una subcolección específica dentro de la colección 'Necesidades' y devuelve todos los documentos como una lista en formato JSON.

### **5.1.5 Obtener Elementos de Patrones**

def get\_PatronesDocInfo():

    datos\_solicitud = request.get\_json()

    doc = datos\_solicitud['Document']

    nameCollection = datos\_solicitud['Name']

    try:

        informacion\_adicional\_ref = db.collection('Patrones').document(doc).collection(nameCollection)

        print(nameCollection)

*Nota: Para acceder al código completo, ir al GitHub:* [*https://lc.cx/uvS-bo*](https://lc.cx/uvS-bo)

Este endpoint permite obtener la información de una subcolección específica dentro de un documento de la colección 'Patrones' en Firestore. Recibe una solicitud POST con los nombres del documento y la subcolección, accede a la subcolección y devuelve todos sus documentos en formato JSON. Si ocurre algún error, devuelve un mensaje de error apropiado.

### **5.1.6 Obtener Elementos de Dominios**

def get\_DominiosLista():

    items = db.collection('Dominios').stream()

    print(items)

    datos\_coleccion = {}

    for i, doc in enumerate(items):

            # Usar el índice como ID comenzando desde 1

            identificador = str(i + 1)

            datos = doc.to\_dict()

            datos["Id"] = doc.id

            datos\_coleccion[identificador] = datos

    return jsonify(datos\_coleccion)

*Nota: Para acceder al código completo, ir al GitHub:* [*https://lc.cx/uvS-bo*](https://lc.cx/uvS-bo)

Este endpoint permite obtener una lista de todos los documentos en la colección 'Dominios' en Firestore. Recupera todos los documentos, los convierte en diccionarios, agrega sus IDs, y devuelve la lista en formato JSON. Esto proporciona una forma estructurada de acceder a todos los documentos de la colección 'Dominios'.

## **5.2 Modulo 2: Desarrollo de Plataforma Web**

El FrontEnd está construido utilizando HTML y JavaScript. La interacción principal del usuario se realiza mediante botones y formularios que permiten consultar y actualizar la información.

### **5.2.1 Configuración Inicial y Carga del DOM**

Esta conexión se realiza principalmente a través de solicitudes HTTP (fetch) que interactúan con los endpoints de la API proporcionada por el servidor Flask, esta línea define la URL base del backend Flask, que se utilizará para todas las solicitudes HTTP.

    fetch(URL+"NecesidadesLista")

        .then(response => {

            if (!response.ok) {

                throw new Error(`Error al obtener el archivo JSON: ${response.status} ${response.statusText}`);

            }

            return response.json();

*Nota: Para acceder al código completo, ir al GitHub:* [*https://lc.cx/FZzLbz*](https://lc.cx/FZzLbz)

* **Variables doc y nameD:** Se utilizan para almacenar información del documento seleccionado.
* **Carga del DOM:** Cuando el DOM se carga completamente, se hace una solicitud fetch al endpoint /NecesidadesLista del backend para obtener una lista de necesidades.
* **Manipulación del DOM:** Se crean botones dinámicamente para cada necesidad obtenida. Cada botón tiene un evento click asociado que llama a la función llamarInformacion con el ID y el título de la necesidad correspondiente.

### **5.2.2 Transformar JSON**

function transformarJson(valor){

    const elementos = valor.split(/\n(?![^\s:]+:)/).filter(Boolean);

    // Crear el objeto JSON en el formato deseado

    const valoracionesJSON = {

    "Resultados": {}

    };

    elementos.forEach((elemento) => {

    const [clave, valor] = elemento.split(':').map(item => item.trim());

*Nota: Para acceder al código completo, ir al GitHub:* [*https://lc.cx/FZzLbz*](https://lc.cx/FZzLbz)

Esta función toma una cadena de texto, la divide en elementos separados por saltos de línea, y crea un objeto JSON a partir de estos elementos. Los valores numéricos se convierten a números, mientras que otros valores permanecen como cadenas.

### **5.2.3 Obtención y Manejo de Información**

function llamarInformacion (documento, name){

    doc = documento;

    nameD = name;

    console.log(documento+name);

    fetch(URL+'DocNecesidadesInfo', {

        method: 'POST',

        headers: {

            'Content-Type': 'application/json'

        },

        body: JSON.stringify({

            Document: documento,

            Name: name

*Nota: Para acceder al código completo, ir al siguiente enlace de GitHub:*

Esta función realiza una solicitud HTTP POST a una URL específica, maneja la respuesta y actualiza la interfaz de usuario en consecuencia. Específicamente diseñada para trabajar con una estructura de datos particular que incluye títulos, definiciones, objetivos, afecciones y cuidados.

Utiliza la API fetch para realizar una solicitud HTTP POST a una URL definida por la variable URL más 'DocNecesidadesInfo'. Los datos de la solicitud se envían en formato JSON, incluyendo documento y name.

Si la solicitud es exitosa, se convierte la respuesta en formato JSON y se maneja la información devuelta. Esto incluye actualizar los valores de ciertos elementos HTML en la página (identificados por sus ID), habilitar o deshabilitar elementos según sea necesario y mostrar información en la consola.

### **5.2.4 Login de Administrador**

            if(data["rol"]=="Administrador"){

                location.href='./administrador.html';

                auth.isAuthenticated = true;

                auth.user = username;

                localStorage.setItem('usuario', data.usuario);

            }else{

                alert("Acceso no autorizado");

            }

        }else{

            alert("Usuario o contraseña incorrecto");

*Nota: Para acceder al código completo, ir al GitHub:* [*https://lc.cx/5s42Mi*](https://lc.cx/5s42Mi)

En este fragmento, se maneja la respuesta del servidor en formato JSON. Si la autenticación es exitosa y el rol del usuario es "Administrador", se redirige a la página administrador.html, y se establece la autenticación como verdadera, el usuario como el nombre de usuario ingresado, y se guarda el usuario en el almacenamiento local.

### **5.2.5 Función de Ocultar y mostrar Información**

PresentarOcultarNece= function(){

mostrarComponente("Necesidades");

    ocultarComponente("Patrones");

    ocultarComponente("Dominios");

    ocultarComponente("Bibliografia");

*Nota: Para acceder al código completo, ir al GitHub:* [*https://lc.cx/mbuOR3*](https://lc.cx/mbuOR3)

Definimos la variable URL que almacena una cadena de texto, que nos ayudara a conectarnos con el backend para la funcionalidad de la presentar y ocultar la información de las necesidades, patrones y dominios, en el portal web administrativo.

## **5.3 Modulo 3: Desarrollo del Chat Bot**

Se desarrolló el chat bot utilizando la API de Gemini, integrándolo tanto en la plataforma web como en la aplicación móvil. Este proceso incluyó la configuración de endpoints en el backend para manejar las consultas de los usuarios y devolver respuestas generadas por el chat bot.

### **5.3.1 Configuración de la IA de Google y Flask**

genai.configure(api\_key="AIzaSyA144dpQmD-S9jCvJhXn2ih8cx2l\_i89FQ")

model = genai.GenerativeModel('gemini-pro')

chat = model.start\_chat(history=[])

chat

app = Flask(\_\_name\_\_)

CORS(app)

*Nota: Para acceder al código completo, ir al GitHub:* [*https://lc.cx/vCWWxY*](https://lc.cx/vCWWxY)

Aquí se configura la clave API para usar los servicios de IA de Google. Se inicializa el modelo de red neuronal 'Gemini-pro' y se inicia una sesión de chat, manteniendo el historial de conversaciones.

Inicia la aplicación Flask y configura CORS para permitir solicitudes desde otros dominios, lo cual es crucial para la interacción entre el FrontEnd y el backend en distintas ubicaciones.

### **5.3.2 Interacción del Chat Bot**

def postChat():

    datos\_solicitud = request.get\_json()

    msj = datos\_solicitud['Mensaje']

        print(chunk.text)

    response = chat.send\_message(msj)

    ConjuntoDatos = ""

*Nota: Para acceder al código completo, ir al GitHub:* [*https://lc.cx/39\_Zmd*](https://lc.cx/39_Zmd)

Este endpoint permite la interacción con un chatbot especializado en enfermería a través de un método POST. Los pasos principales incluyen:

* Recepción de un mensaje JSON del cliente.
* Configuración inicial del chatbot para definir su comportamiento y ámbito de conocimiento.
* Envío del mensaje del cliente al chatbot y procesamiento de la respuesta.
* Devolución de la respuesta del chatbot al cliente en formato JSON.

El servidor Flask se configura para ejecutarse en el puerto 5000, y la ruta /Chat/Chatbot está diseñada específicamente para manejar solicitudes POST, facilitando una comunicación interactiva y dinámica con el chatbot.

### **5.3.3 Almacenamiento de Historial del Chat**

 const loadInitialMessages = async () => {

    const stored = await AsyncStorage.getItem('userData');

    const userData = JSON.parse(stored);

    setData(userData.usuario);

    console.log(Data);

*Nota: Para acceder al código completo, ir al GitHub:* [*https://lc.cx/39\_Zmd*](https://lc.cx/39_Zmd)

Esta función carga los datos iniciales del usuario cuando el componente el ChatScreen se monta. Estos datos pueden ser necesarios para realizar ciertas operaciones, como enviar mensajes al servidor o guardar el historial de chat en el almacenamiento local.

messages: Almacena los mensajes del chat y Data: Almacena información del usuario actual.

* **AsyncStorage.getItem('userData'):** Recupera los datos del usuario almacenados en AsyncStorage bajo la clave 'userData'.
* **JSON.parse(stored):** Convierte los datos del usuario de formato JSON a un objeto JavaScript.
* **setData(userData.usuario):** Establece los datos del usuario en el estado Data.
* **console.log(Data):** Imprime los datos del usuario en la consola
* if (storedMessages) {
* // Si hay mensajes almacenados
* setMessages(JSON.parse(storedMessages));
* console.log(JSON.parse(storedMessages));
* } else {

*Nota: Para acceder al código completo, ir al GitHub:* [*https://lc.cx/39\_Zmd*](https://lc.cx/39_Zmd)

Esta parte del código intenta cargar el historial de mensajes del usuario actual y establecerlo como estado del componente ChatScreen. Si no hay mensajes almacenados, se establecerá un estado predeterminado o vacío.

 const sendUserMessage = async (text) => {

    const data = { text };

    try {

      // Realizar el POST al servidor con el texto del usuario

      const response = await axios.post(`${BASE\_URL}/Chat/Chatbot`, { Mensaje: data });

*Nota: Para acceder al código completo, ir al GitHub:* [*https://lc.cx/39\_Zmd*](https://lc.cx/39_Zmd)

La función sendUserMessage es responsable de enviar mensajes del usuario al servidor y manejar las respuestas del servidor. Se integra con Axios para realizar solicitudes HTTP y proporciona un mecanismo para manejar errores de manera adecuada.

const saveMessagesToStorage = async (messagesToSave) => {

    try {

      // Guarda los mensajes en AsyncStorage como una cadena JSON

      const valor = AsyncStorage.setItem('chat\_history'+Data, JSON.stringify(messagesToSave));

      console.log(valor);

*Nota: Para acceder al código completo, ir al GitHub:* [*https://lc.cx/39\_Zmd*](https://lc.cx/39_Zmd)

La función saveMessagesToStorage se encarga de guardar los mensajes proporcionados en el almacenamiento local del dispositivo. Utiliza AsyncStorage para esta tarea, convirtiendo el array de mensajes en una cadena JSON antes de guardarlos. También proporciona manejo de errores para garantizar una ejecución segura

## **5.4 Modulo 4: Desarrollo del Aplicativo Móvil**

La aplicación móvil desarrollada en React Native ofrece una plataforma versátil y eficiente para la creación de aplicaciones móviles multiplataforma. React Native permite a los desarrolladores utilizar un único código base para construir aplicaciones para iOS y Android, lo que facilita el desarrollo y la mantenibilidad del proyecto.

La aplicación se conecta a un backend para obtener y enviar datos, lo que permite la sincronización de información entre el dispositivo móvil y el servidor. La comunicación con el backend se realiza mediante solicitudes HTTP utilizando librerías como Fetch API.

### **5.4.1 Navegación**

import react, { useContext } from "react";

import { Image, View, Text,TouchableOpacity } from 'react-native';

import { NavigationContainer } from '@react-navigation/native';

import { createNativeStackNavigator } from '@react-navigation/native-stack';

*Nota: Para acceder al código completo, ir al GitHub:* [*https://lc.cx/UOmq\_C*](https://lc.cx/UOmq_C)

Las importaciones son esenciales para incluir módulos y componentes necesarios en el archivo. En este caso, se importan módulos como react, useContext, componentes de React Native como Image, View, Text, TouchableOpacity, así como varios módulos de navegación de React Navigation.

Se crean los componentes de navegación utilizando createNativeStackNavigator para la navegación de tipo stack y createBottomTabNavigator para la navegación de tipo pestañas

      <Tab.Screen name="Necesidades" component={NecesidadesScreen} options={{headerShown:false,

        tabBarIcon: ({ focused, color, size }) => (

          <Image

            source={focused ? require('./img/nece\_1.png') :

*Nota: Para acceder al código completo, ir al GitHub:* [*https://lc.cx/UOmq\_C*](https://lc.cx/UOmq_C)

Se define el componente TabNavigator que contiene la navegación de tipo pestañas utilizando Tab.Navigator, se especifican las configuraciones de estilo y las pantallas de pestañas dentro de Tab.Navigator.

### **5.4.2 Registro**

onst Registro = ({route}) => {

    const navigation = useNavigation();

    const [nombre, setNombres] = useState(null);

    const [pais, setPais] = useState(null);

    const [ciudad, setCiudad] = useState(null);

*Nota: Para acceder al código completo, ir al GitHub:* [*https://lc.cx/vMePjY*](https://lc.cx/vMePjY)

Este fragmento de código define un componente de registro que utiliza estados locales para almacenar los datos ingresados por el usuario y el contexto de autenticación para realizar el registro en el backend cuando el usuario envía el formulario.

                        placeholder="Ingrese su nombre"

                        onChangeText={text => setNombres(text)}

                        />

                </View>

                <View style={styles.txtInput}>

                    <View style={styles.IconText}>

                        <Text style={styles.TextoUsuario} >Pais</Text>

*Nota: Para acceder al código completo, ir al GitHub:* [*https://lc.cx/vMePjY*](https://lc.cx/vMePjY)

Este fragmento de código nos da indica como se está desarrollando la parte del FrontEnd de la vista de registro de la app móvil, en la que se ingresa los campos requeridos para que el usuario pueda registrase y así obtener los datos.

### **5.4.3 Autentificación**

 const login = async (email, password) =>{  
setIsLoading(true);  
        axios.post(`${BASE\_URL}/verificar\_usuario`,{  
            email,  
            password  
        }).then(res => {           
            let userInfo = res.data;

*Nota: Para acceder al código completo, ir al GitHub:* [*https://lc.cx/kGAQdA*](https://lc.cx/kGAQdA)

Se Utiliza axios para realizar una solicitud POST al servidor con las credenciales de usuario (email y password). Luego, procesa la respuesta del servidor, actualiza el estado del usuario (userInfo) y guarda los datos de usuario en el almacenamiento local utilizando AsyncStorage. Si la autenticación es exitosa (indicada por userInfo.status), redirige al usuario a la pantalla de contenido (Contenido).

 const checkUserAuthentication = async () =>{

        try {

            const userData = await AsyncStorage.getItem('userData');

            if (userData) {

                //console.log('Usuario autenticado');

            } else {

                RootNavigation.navigate('Login');

                //console.log('Usuario no autenticado');

*Nota: Para acceder al código completo, ir al GitHub:* [*https://lc.cx/kGAQdA*](https://lc.cx/kGAQdA)

Esta función se encarga de verificar si hay datos de usuario almacenados en el dispositivo. Utiliza AsyncStorage.getItem para obtener los datos de usuario. Si hay datos de usuario, se asume que el usuario está autenticado y no se realiza ninguna acción. Si no hay datos de usuario, se redirige al usuario a la pantalla de inicio de sesión (Login)

### **5.4.4 Muestra de Información**

const DesNecesidades = ({route}) => {

useState(false);

    const [mostrarInformacionAfecciones, setMostrarInformacionAfecciones] = useState(false);

*Nota: Para acceder al código completo, ir al GitHub:* [*https://lc.cx/BYYCyB*](https://lc.cx/BYYCyB)

El useState se utiliza para a manejar el estado local del componente. Esto incluye estados para almacenar datos (data y datosBibliografia), controlar la visibilidad de la información (mostrarInformacion, mostrarInformacionAfecciones, mostrarInformacionCuidados, mostrarInformacionBiblio), y almacenar resultados de afecciones, cuidados y bibliografía (resultAfecciones, resultadosCuidados, resultadosBiblio).

axios.post(`${BASE\_URL}/DocNecesidadesInfo`, {

            Name:Document,

            Document:Id

            });

            const token = AsyncStorage.setItem('AlmacenNecesidades'+Id, JSON.stringify(response.data));

            setData(response.data);

*Nota: Para acceder al código completo, ir al GitHub:* [*https://lc.cx/Gnli8j*](https://lc.cx/Gnli8j)

Esta función realiza una solicitud POST a la API para obtener datos relacionados con las necesidades, almacena estos datos localmente y los actualiza en el estado local del componente, los datos obtenidos en la respuesta se almacenan localmente utilizando AsyncStorage, Los datos también se establecen en el estado local utilizando la función setData.