TECNOLÓGICO NACIONAL DE MÉXICO

Instituto Tecnológico de Mexicali

Ing. Sistemas Computacionales

Desarrollo de Aplicaciones Web

Docente: Valenzuela Bogarin Jose Ramon

“Proyecto Mind Care – Sprint 3”

Integrantes:
Rodriguez Herrera Maria Fernanda – 21490574
Hernández López Jose Carlos – 22150084
Favila Sánchez Angel Jair – 21490074

Mexicali, B.C. **07-Nov-2025**

Índice

Problemática

La salud mental es un tema de suma importancia, ya que afecta en el día a día de las personas, tanto en sus relaciones sociales como personales.

Es por ello que Mind Care busca que las personas puedan agendar sesiones con un psicólogo, para manejar sus citas desde la comodidad de su casa.

Objetivo del proyecto

Crear una aplicación web en la cual los usuarios puedan agendar citas para sesiones psicológicas, enviándoles un recordatorio 24 horas antes a su cita para confirmarla o cancelarla.

Historias de usuario

Usuario	• Como paciente nuevo, quiero poder registrarme para tener una cuenta y posteriormente iniciar sesión • Como paciente, quiero ver los días y horas disponibles para agendar mi cita. • Como paciente, quiero agendar una cita seleccionando fecha y hora. • Como paciente, quiero visualizar mis citas agendadas. • Como paciente, quiero que se me notifique 24 horas un recordatorio de mi cita por medio de email. • Como paciente, quiero poder confirmar y cancelar mis citas por medio del recordatorio.
Psicólogo	• Como psicólogo nuevo, quiero poder registrarme para tener una cuenta y posteriormente iniciar sesión • Como psicólogo, quiero visualizar las citas agendadas del día. • Como psicólogo, quiero crear notas para tener un historial de mis pacientes. • Como psicólogo, quiero ver el historial de mis pacientes. • Como psicólogo, quiero marcar días inhábiles.

Módulos funcionales y no funcionales

• **Funcionales - Pacientes:**

- RF01: Registro

- RF02: Inicio de sesión

- RF03: Autenticación

- RF04: Agendar citas (Una por dia)

- RF05: Aceptar citas (por correo)

- RF06: Cancelar citas (por correo)

- RF07: Recordatorio de cita 24 horas antes (por correo)

- RF08: Historial de citas

- RF09: Calendario con disponibilidad de citas (día y hora)

- RF10: Correo notificando la cancelación de una cita, por día inhábil del psicólogo

• **Funcionales - Psicólogo:**

- RF01: Registro de psicólogo

- RF02: Inicio de sesión

- RF03: Autenticación

- RF04: Visualización de citas (Calendario)

- RF05: Historial de citas

- RF06: Creación de notas para el historial del paciente

- RF07: Modificar notas

- RF08: Crear de días inhábiles, y cancelación de citas en ese mismo día.

- RF09: Ver expediente de mis pacientes.

• No funcionales:

- RNF01: Interfaz intuitiva, accesible y rápida de entender

- RNF02: Privacidad de acceso a notas del psicólogo

- RNF03: Restricciones de horario para sistema de recordatorios

Tipo de arquitectura

Monolítica – Multicapa

1-. Monolítica

- ¿Qué es? Arquitectura donde toda la aplicación se desarrolla, construye y despliega como una sola unidad, incluyendo la interfaz de usuario, la lógica de negocio y el acceso a datos, todo en un solo bloque de ejecución.

- Responsabilidad: Centraliza toda la funcionalidad en un solo sistema, acoplando las capas.

- Ejemplo proyecto: Dentro de la aplicación se harían todas las funcionalidades (inicio de sesión, envío de correos, etc.), a la vez que la base de datos se compartiría por todos los módulos.

2-. Multicapa:

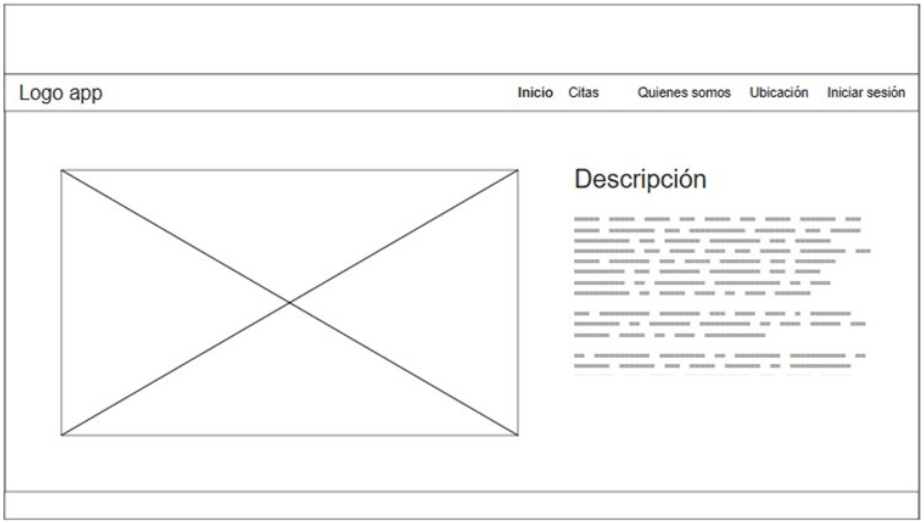
- ¿Qué es? Divide la aplicación en capas lógicas independientes, teniendo cada una su responsabilidad específica.

- Responsabilidad: Separar las responsabilidades para que cada capa se centre en una tarea específica, mejorando la mantenibilidad, escalabilidad y reutilización.

- Ejemplo proyecto: Separar la aplicación en capas de presentación, negocio y datos; para el proceso del manejo de sesiones.

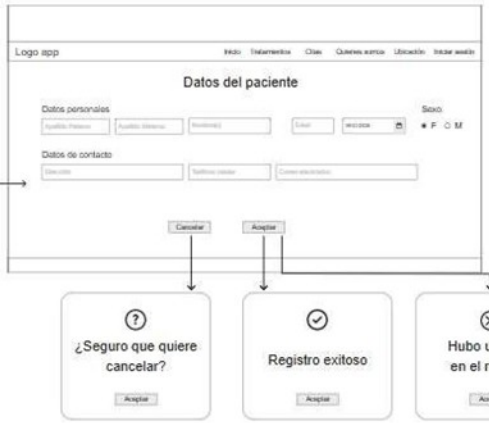
Wireframes

Página Inicio

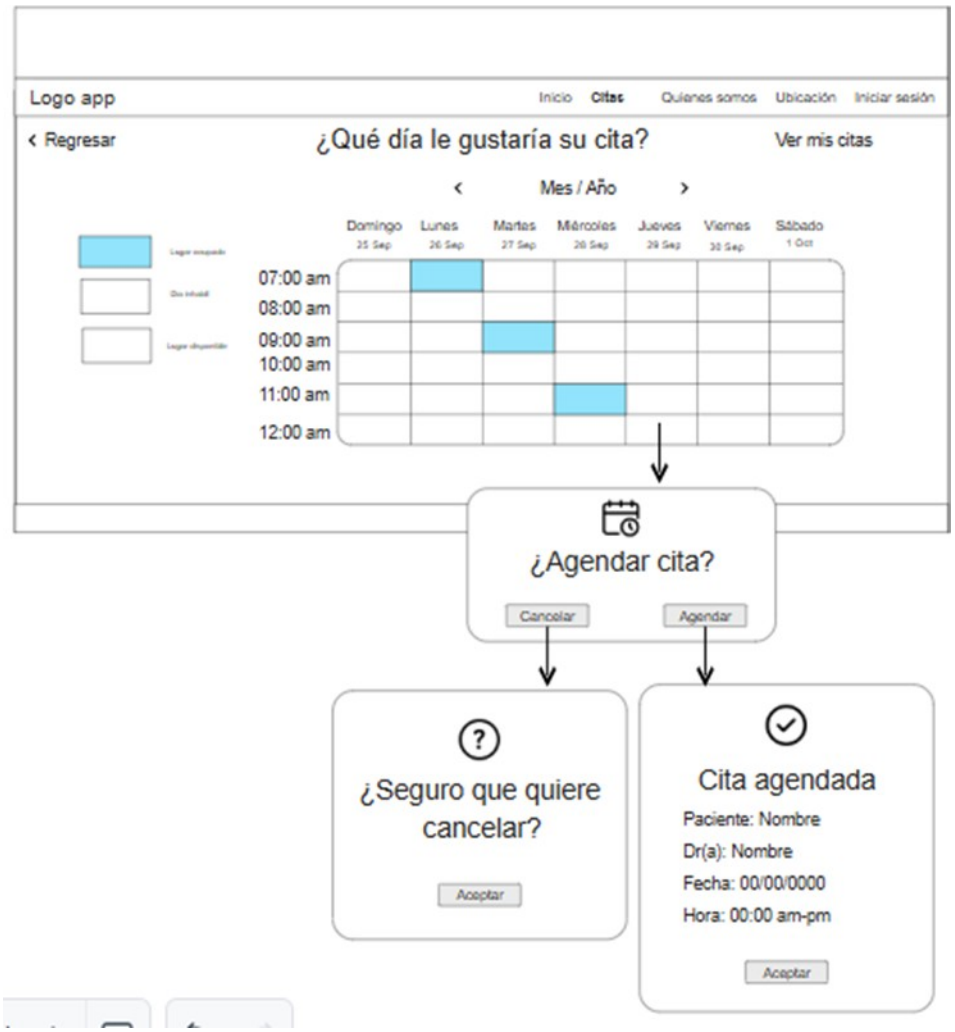


Iniciar sesión

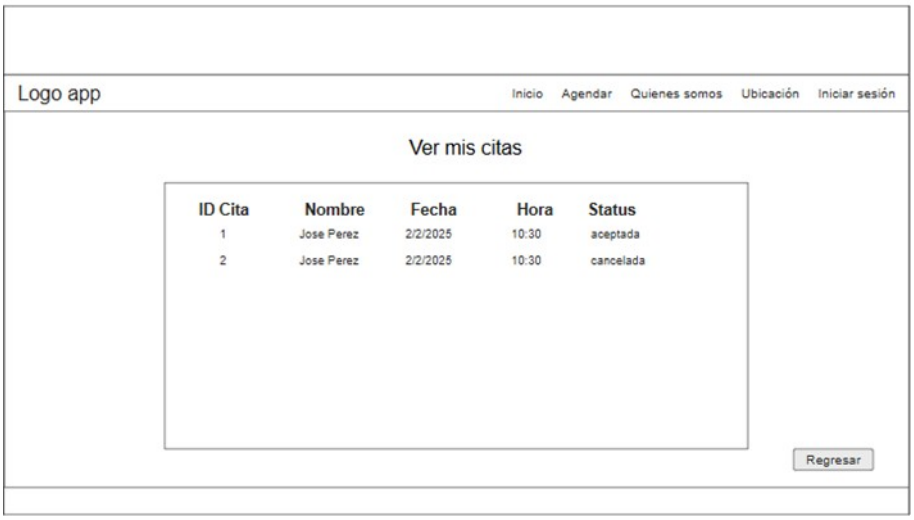
Registro



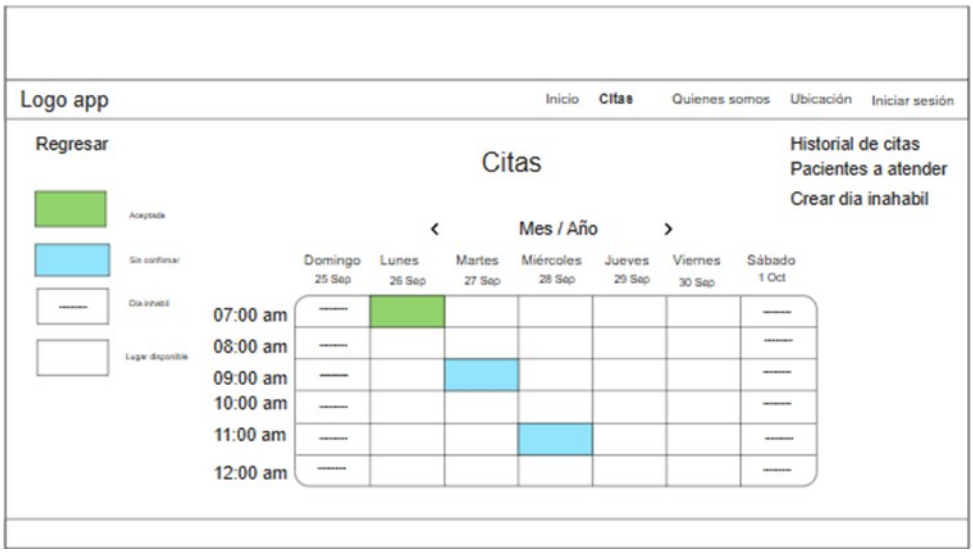
Agendar cita - Paciente



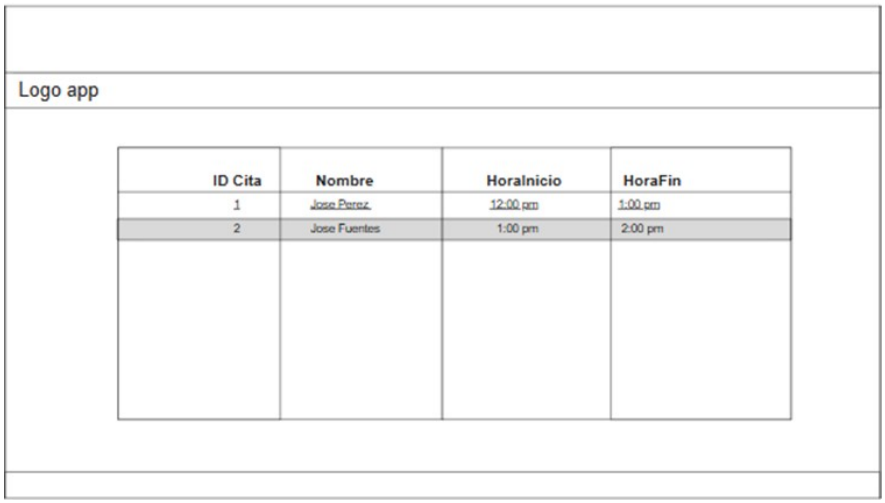
Mis citas - Paciente



Visualizar citas - Psicólogo



"Recepcionista virtual"



Plan de Sprints

*** Sprint 1: 29 sep → 10 oct**

Entregables:

• Menú de navegación (Inicio, Citas, Ubicación, Quienes somos, Iniciar sesión).

• Páginas estáticas: Inicio, Quienes somos, Ubicación, Iniciar sesión y Registro.

• Diseño de la base de datos.

*** Sprint 2: 13 oct → 24 oct**

Entregables:

• Creación de la base de datos.

• Registro e inicio de sesión funcionales.

*** Sprint 3: 27 oct → 7 nov**

Entregables:

• Vista de Calendario para el usuario.

• Función de agendar cita (Sin correo de recordatorio).

• Estados de cita: Pendiente, Aceptada (Si es el mismo día).

• Historial de citas: Solamente las aceptadas, sin notas por parte del psicólogo.

*** Sprint 4: 10 nov → 21 nov**

Entregables:

• Vista del calendario para el psicólogo (Aceptadas el mismo día, pendientes).

• Crear días inhábiles (Cancela las citas existentes de ese día).

• Historial de citas aceptadas el mismo día o canceladas por día inhábil.

*** Sprint 5: 24 nov → 5 dic**

Entregables:

• Configuración de servicio de correo en Flask.

• Envío automático para confirmaciones/cancelaciones

• Envío de correo de cancelación de cita por día inhábil del psicólogo.

• Visualización de expediente de pacientes.

• Cancelación y confirmación desde correo (Actualiza la BD y el calendario)

• Tabla con los pacientes a atender

• Deployment

Criterios aceptables

- Permitir a los pacientes y psicólogo registrarse para tener una cuenta y, posteriormente iniciar sesión con su cuenta para utilizar la aplicación.

- Permitir a los pacientes crear y visualizar sus citas.

- Implementar un sistema de recordatorios por email de 24 horas antes para las citas.

- Permitir al psicólogo cancelar citas y marcar días inhábiles.

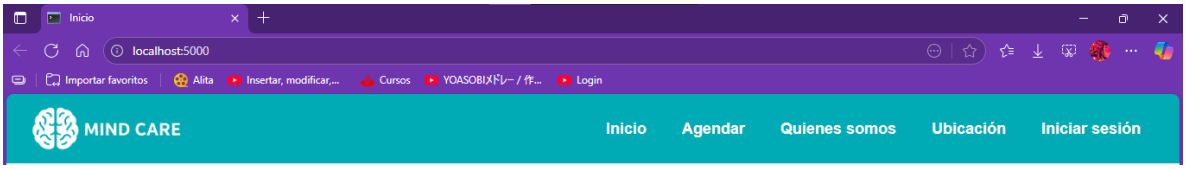
- Permitir al psicólogo crear y modificar notas para el historial de sus pacientes.

- Permitir a los pacientes y psicólogo cerrar la sesión de su cuenta.

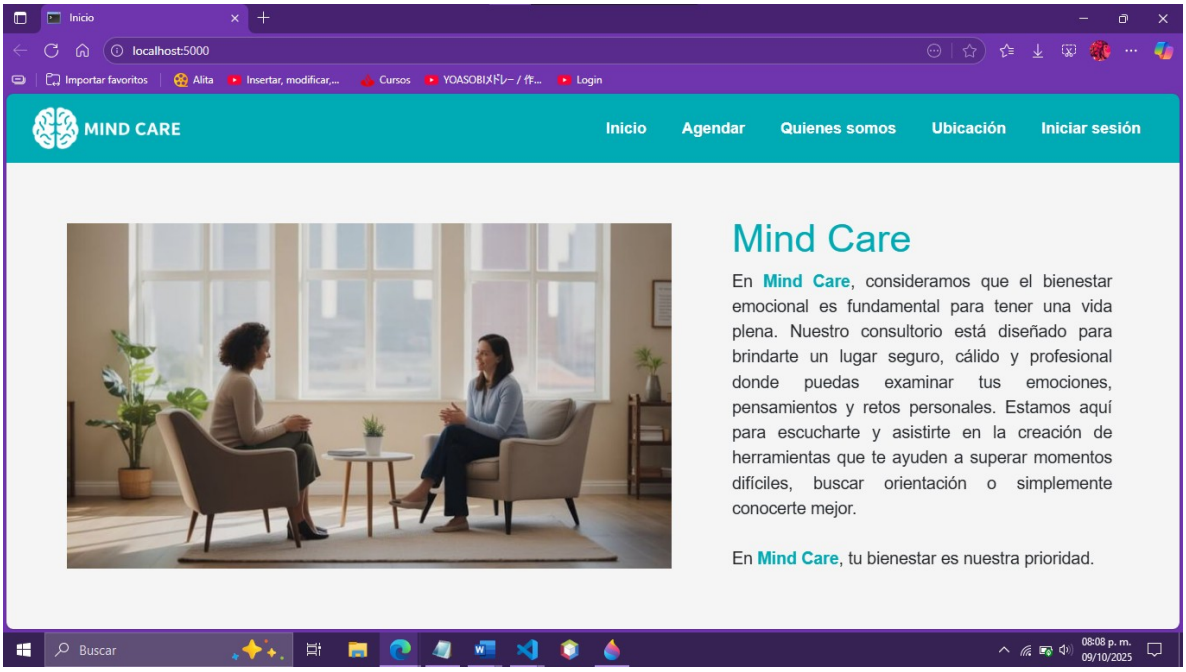
Sprint 1: 29 sep - 10 oct

Entregables:

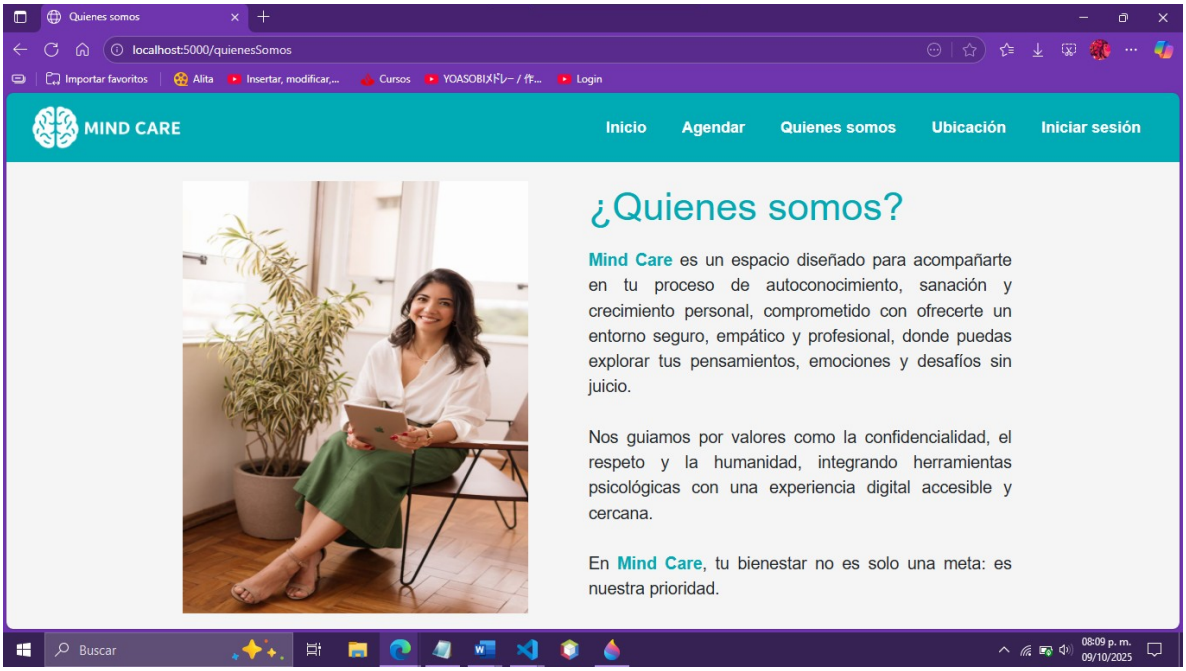
• Menú de navegación (Inicio, Citas, Ubicación, Quienes somos, Iniciar sesión).



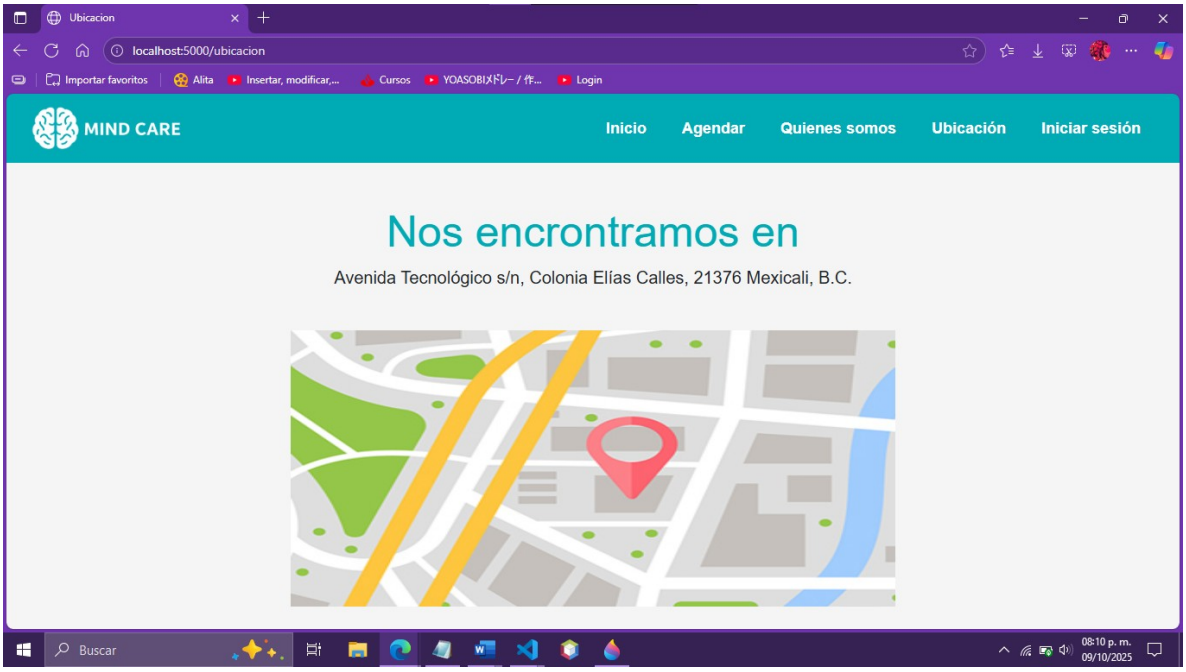
• Páginas estáticas: Inicio



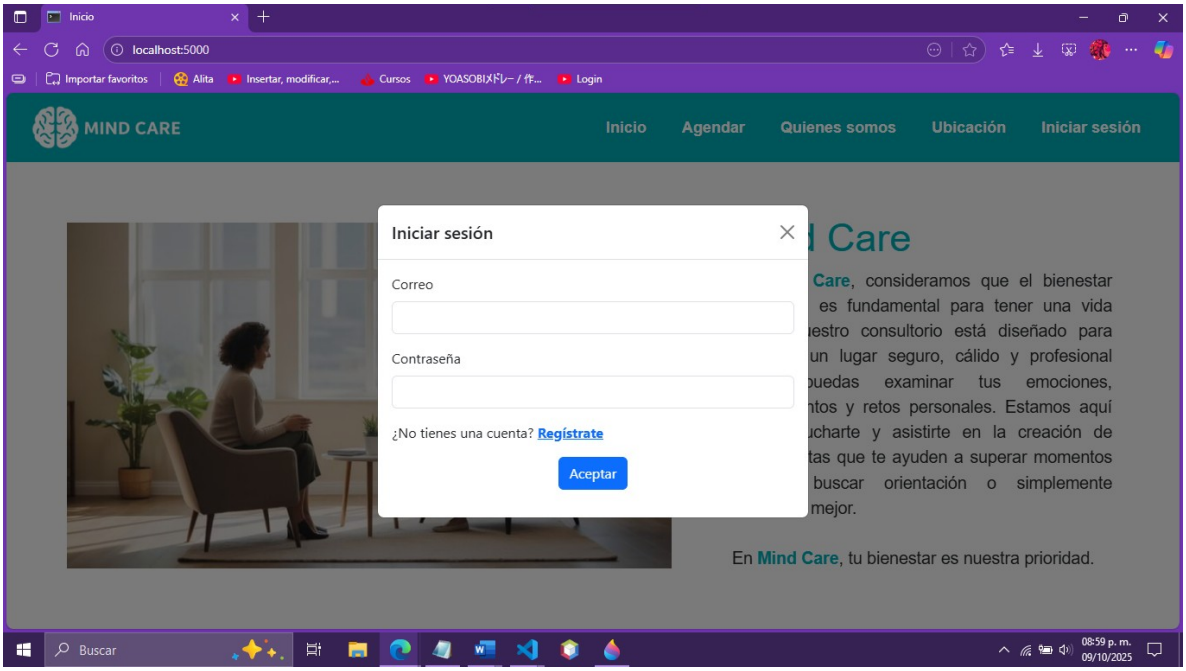
• Páginas estáticas: Quienes somos



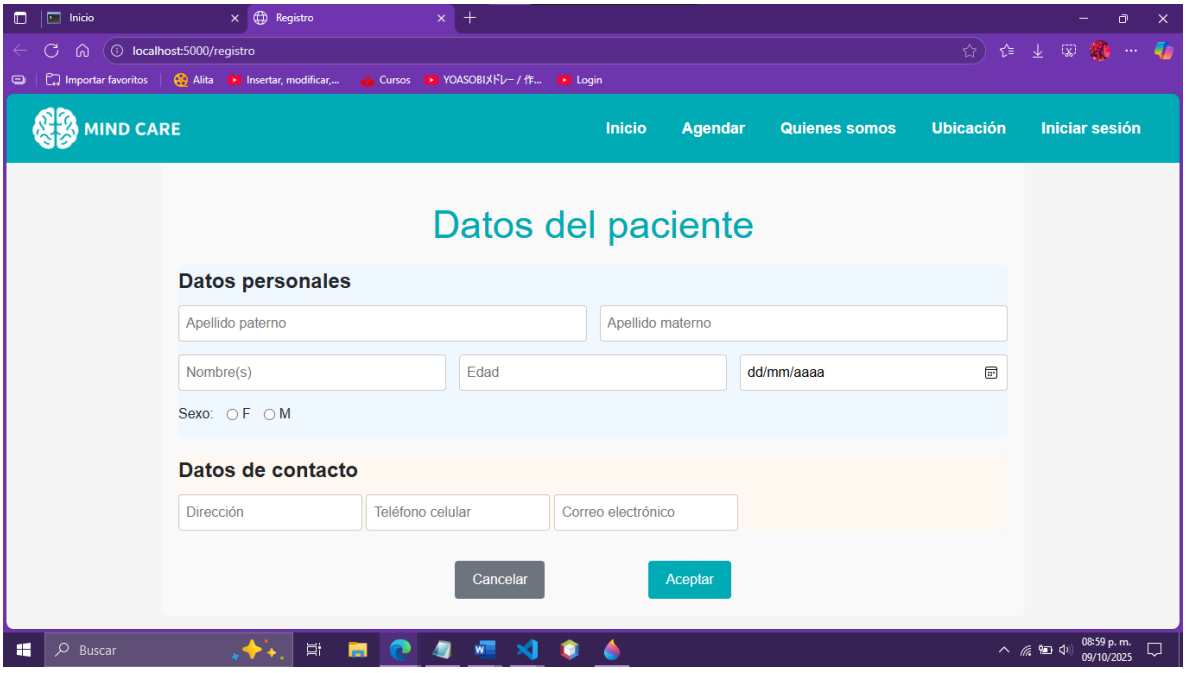
• Páginas estáticas: Ubicación



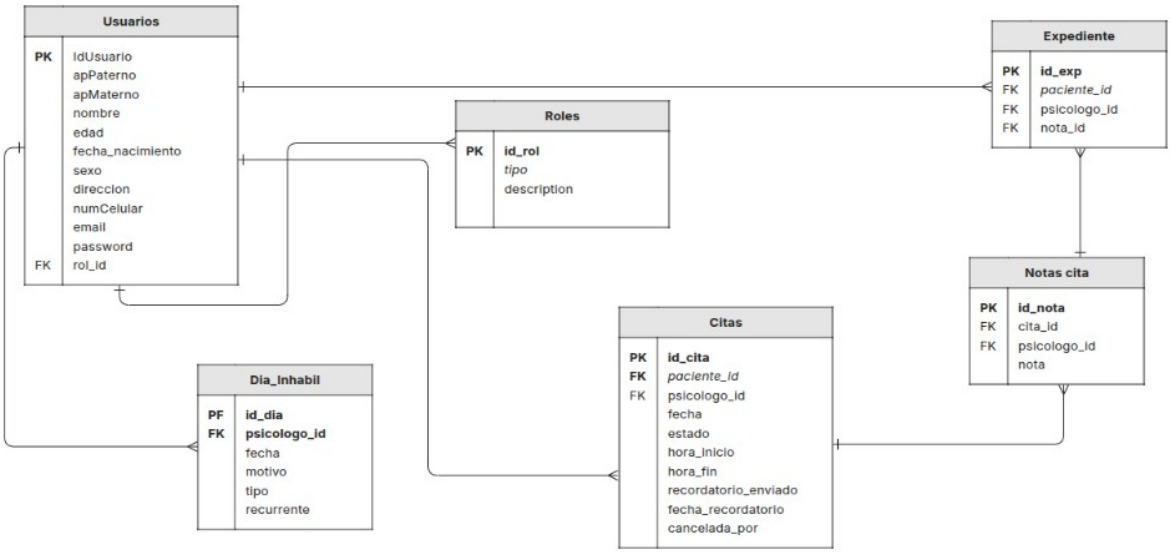
• Páginas estáticas: Iniciar sesión



• Páginas estáticas: Registro

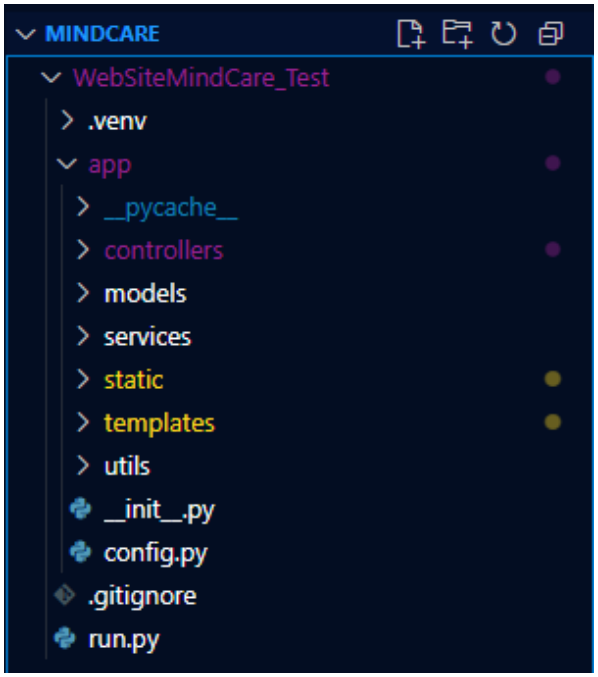


• Diseño de la base de datos.



Manejo de arquitectura Monolítica-Multicapa

Como se nos pidió trabajar nuestro proyecto en base a las arquitecturas monolítica y multicapa, se dividió el proyecto en las siguientes carpetas: ***controllers, models, services, static, templates*** y ***utils,*** en las cuales de momento se han trabajado con ***controllers, static*** y ***templates***, ya que se tienen los archivos ***html, css*** e ***imágenes*** utilizadas en la aplicación, así como la definición de rutas para la navegación de la misma:



En nuestro archivo ***main_routes.py*** definimos las rutas para la navegación de nuestra aplicación, las cuales dirigen al usuario a las páginas de inicio, quienes somos, etc. La herramienta de ***Blueprint*** sirve para organizar la aplicación al dividirla en componentes más pequeños y manejables:

```python
from flask import Blueprint, render_template    No se h

main_bp = Blueprint ('main_bp',__name__)

#Rutas para archivos principales
@main_bp.route('/')
def inicio():
    return render_template('public/inicio.html')

@main_bp.route('/agendar')
def citas():
    return render_template('public/agendar.html')

@main_bp.route('/quienesSomos')
def quienesSomos():
    return render_template('public/quienesSomos.html')

@main_bp.route('/ubicacion')
def ubicacion():
    return render_template('public/ubicacion.html')

@main_bp.route('/login')
def login():
    return render_template('public/login.html')

@main_bp.route('/registro')
def registro():
    return render_template('public/registro.html')
```

En nuestra carpeta de ***templates*** (utilizada y reconocida por ***Flask*** para renderizar los archivos html) ***/public*** creamos un archivo llamado ***menu.html***, el cual contiene la estructura y links de navegación para la aplicación. El propósito de esta es reutilizar el archivo mandándolo a llamar en las demás páginas para ahorrar código y aprovechar el motor de plantilla de ***Flask*** que es ***Jinja2:***

menu.html M

WebSiteMindCare_Test > app > templates > public > menu.html > ...

```html
<nav class="navbar navbar-expand-lg bg-body-tertiary">
    <div class="container-fluid">
        <div class="logo">
            <a href="{{ url_for('main_bp.inicio') }}">
                <img src="{{ url_for('static', filename='img/logoApp.png') }}">
            </a>
        </div>
        <div class="collapse navbar-collapse" id="navbarNav">
            <ul class="navbar-nav">
                <li class="nav-item">
                    <a class="nav-link" href="{{ url_for('main_bp.inicio') }}">Inicio</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link"  data-bs-toggle="modal" data-bs-target="#loginModal">Agendar</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="{{ url_for('main_bp.quienesSomos') }}">Quienes somos</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="{{ url_for('main_bp.ubicacion') }}">Ubicación</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" data-bs-toggle="modal" data-bs-target="#loginModal">Iniciar sesión</a>
                </li>
            </ul>
        </div>
    </div>
</nav>
```

De igual manera, en nuestra carpeta ***templates/public*** creamos un archivo llamado ***base.html***, el cual contiene los links a ***Bootstrap*** y al ***style.css*** para facilitar los diseños, así como incluir al archivo ***menu.html*** para reutilizarlo, y en este mismo incluye ***login_modal.html*** que contiene la estructura de nuestro Pop up del login. Se utilizó ***include*** ya que sirve para reutilizar fragmentos de código HTML, como el menú, mediante el motor de plantillas ***Jinja2*** (como se mencionó anteriormente):

base.html

WebSiteMindCare_Test > app > templates > base.html > ...

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title> {% block title %} Pagina base {% endblock %} </title>
     <!--Bootstrap-->
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css" rel="stylesheet">
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/bootstrap.bundle.min.js"></script>
    <!--Style-->
    <link rel="stylesheet" href="{{ url_for('static', filename='css/style.css') }}">
    {% block style %}

    {% endblock %}

</head>
<body>
    {% include 'public/menu.html' %}
    {% block content %}

    {% endblock %}
    {% include 'components/login_modal.html' %}
</body>
</html>
```

Parala estructura y diseño de nuestras páginas se utilizó HTML y CSS, así como estructuras de control que ayudan a controlar el flujo de ejecución del contenido que se renderiza, permitiendo mostrar los elementos especificados. Así se manejó la estructura para las demás pantallas (quienes somos, ubicación):

inicio.html

WebSiteMindCare_Test > app > templates > public > inicio.html > ...

```
{% block title %} Inicio {% endblock %}

{% block style %}
    <link rel="stylesheet" href="{{ url_for('static', filename='css/style_inicio.css') }}">
{% endblock %}


{% extends 'base.html' %}
{% block content %}

    <div class="content">
        <img src="{{ url_for('static', filename='img/psicologoInicio.jpg') }}">
        <div class="text">
            <h1>Mind Care</h1>
            <p>En <strong>Mind Care</strong>, consideramos que el bienestar emocional es
            fundamental para tener una vida plena. Nuestro consultorio está diseñado
            para brindarte un lugar seguro, cálido y profesional donde puedas examinar
            tus emociones, pensamientos y retos personales. Estamos aquí para escucharte
            y asistirte en la creación de herramientas que te ayuden a superar momentos
            difíciles, buscar orientación o simplemente conocerte mejor.
            <br><br>
            En <strong>Mind Care</strong>, tu bienestar es nuestra prioridad.</p>
        </div>
    </div>
{% endblock %}
```

ubicacion.html

WebSiteMindCare_Test > app > templates > public > ubicacion.html > ...

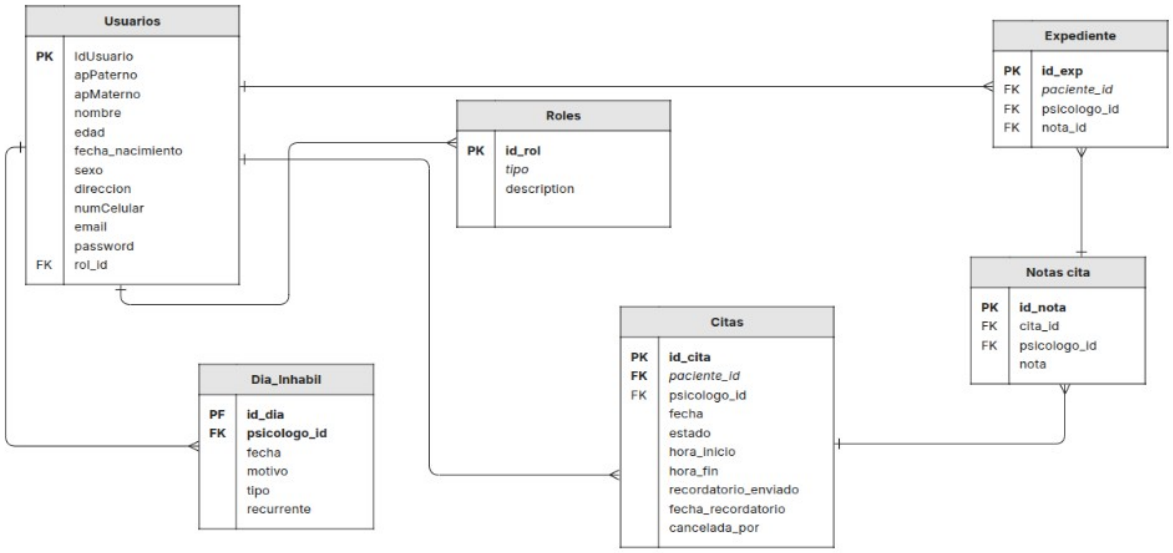
```
{% block title %} Ubicacion {% endblock %}

{% block style %}
    <!--Style-->
    <link rel="stylesheet" href="{{ url_for('static', filename='css/style_ubicacion.css') }}">
{% endblock %}

{% extends 'base.html' %}
{% block content %}
    <div class="content">
        <h1>Nos encrontramos en</h1>
        <p>Avenida Tecnológico s/n, Colonia Elías Calles, 21376 Mexicali, B.C.</p>
        <img src="{{ url_for('static', filename='img/ubicacion.jpg') }}">
    </div>
{% endblock %}
```

quienesSomos.html M

WebSiteMindCare_Test > app > templates > public > quienesSomos.html > ...

```
{% block title %} Quienes somos {% endblock %}
{% block style %}
    <!--Style-->
    <link rel="stylesheet" href="{{ url_for('static', filename='css/style_quienesSomos.css') }}">
{% endblock %}

{% extends 'base.html' %}
{% block content %}
    <div class="content">
        <img src="{{ url_for('static', filename='img/psicologa.jpg') }}">
        <div class="text">
            <h1>¿Quienes somos?</h1>
            <p><strong>Mind Care</strong> es un espacio diseñado para acompañarte
            en tu proceso de autoconocimiento, sanación y crecimiento personal,
            comprometido con ofrecerte un entorno seguro, empático y profesional,
            donde puedas explorar tus pensamientos, emociones y desafíos sin juicio.
            <br><br>
            Nos guiamos por valores como la confidencialidad, el respeto y la
            humanidad, integrando herramientas psicológicas con una experiencia
            digital accesible y cercana.
            <br><br>
            En <strong>Mind Care</strong>, tu bienestar no es solo una meta: es nuestra prioridad.</p>
        </div>
    </div>
{% endblock %}
```

Y por la parte del diseño de nuestra base de datos, se contemplaron las tablas para ***usuarios, roles, dia_inhabil, citas, notas_citas*** y ***expediente***:



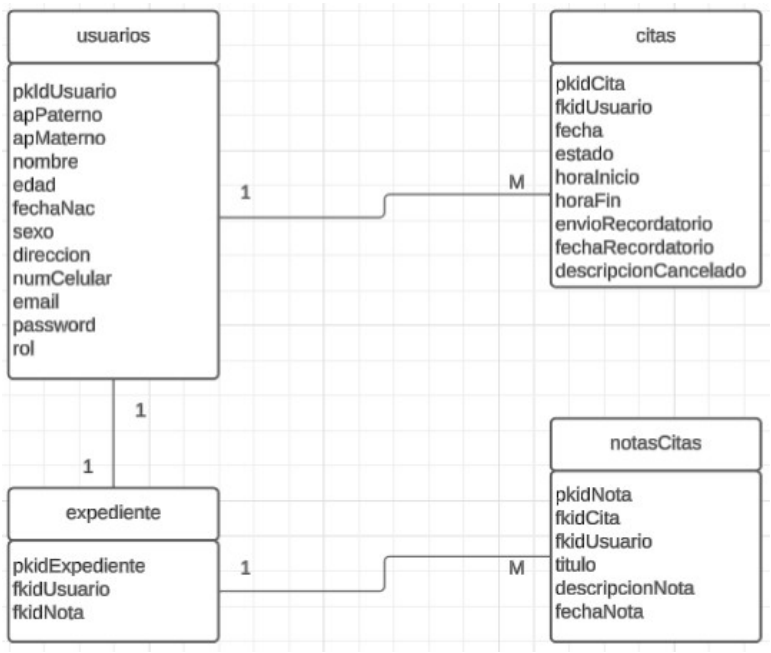
Debido al problema de redundancia con las relaciones en las tablas de **Usuarios, Citas, Notas cita** y **expediente** detectado al presentar el sprint 1, el diseño de la base de datos se modificará y se presentará en el segundo sprint (24 de octubre del 2025), ya que se necesita realizar un mejor análisis en la estructura de esta misma.

Sprint 2: 13 oct - 24 oct

Entregables:

• Creación de la base de datos.

En este segundo sprint, de acuerdo a la observación hecha por el profesor en el sprint pasado acerca del diseño de la base de datos, se realizó un nuevo análisis acerca de las relaciones y campos de las tablas con ayuda de nuestro docente, llegando a la siguiente conclusión:



Se decidió quitar las tablas de ***roles*** y ***dia_inhabil***, dejándolas como campos en las tablas de ***usuario (rol)*** y ***citas (dia_inhabil)***, llegando a una mayor limpieza en nuestro diseño de base de datos. Por lo que, para empezar a trabajar en nuestro proyecto, en nuestro archivo ***requirements.txt*** se agregó lo siguiente:

```
# --- Base de datos ---
Flask-SQLAlchemy==3.1.1
psycopg2-binary==2.9.9
```

Las cuales son librerías para trabajar con bases de datos en ***Python.***

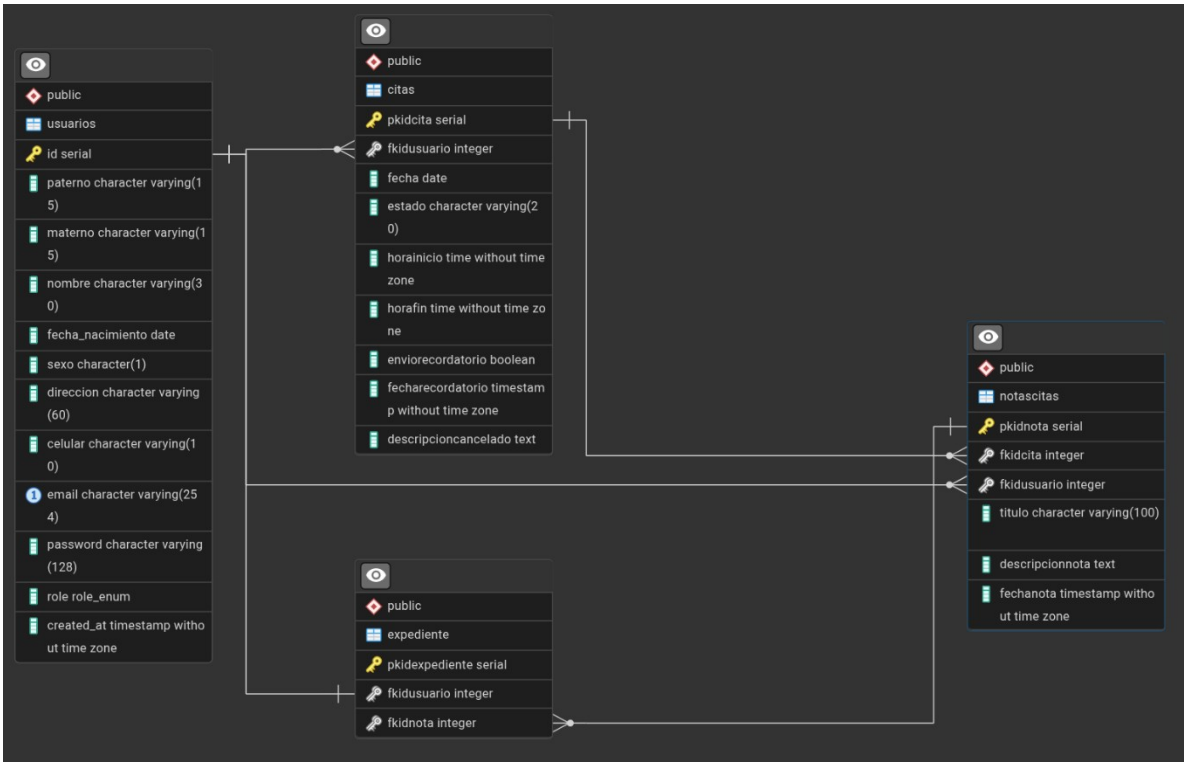
Posteriormente, se creó un archivo llamado ***config.py*** para establecer la conexión con nuestra base de datos, la cual contiene la ***SQLALCHEMY_DATABASE_URI*** que especifica el usuario, contraseña, puerto y base de datos a la que se va a conectar. Seguido de esto se tiene ***SQLALCHEMY_TRACK_MODIFICATIONS*** en false, lo cual sirve para ahorrar memoria y mejorar el rendimiento, y por último ***SECRET_KEY*** que es para firmar cookies y sesiones en nuestra app.

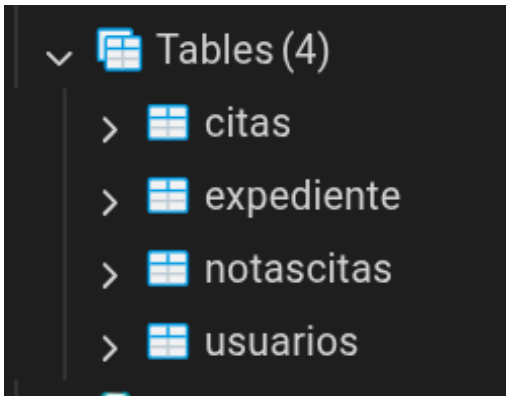
```python
import os

class Config:
    SQLALCHEMY_DATABASE_URI = 'postgresql://user_postgres:password_postgres@localhost:5433/db_postgres'
    SQLALCHEMY_TRACK_MODIFICATIONS = False
    SECRET_KEY = 'mind_care_project'
```

Una vez teniendo esto listo, se procedió a realizar un contenedor llamado ***docker-compose.yml*** para levantar una instancia de ***postgres*** y poder trabajar con nuestra base de datos, teniendo definidos datos como la imagen (postgres 16), el nombre del contenedor, así como el usuario, contraseña, nombre de la base de datos y el puerto. De igual manera, se estableció una imagen para usar ***pgadmin***, la cual es la interfaz gráfica en donde trabajaremos de una mejor manera nuestra base de datos, teniendo establecidos el ***email*** y ***password*** con los que iniciaremos para conectarnos a nuestro servidor.

```yaml
version: '3.8'

services:
  db:
    image: postgres:16
    container_name: postgres_db
    restart: always
    environment:
      POSTGRES_USER: user_postgres
      POSTGRES_PASSWORD: password_postgres
      POSTGRES_DB: db_postgres
    ports:
      - "5433:5432"
    volumes:
      - ./postgres_data:/var/lib/postgresql/data

  pgadmin:
    image: dpage/pgadmin4
    container_name: pgadmin
    restart: always
    environment:
      PGADMIN_DEFAULT_EMAIL: admin@admin.com
      PGADMIN_DEFAULT_PASSWORD: admin
    ports:
      - "5050:80"
    depends_on:
      - db
```

Entonces, para ingresar a ***pgadmin***, en nuestro navegador ingresamos ***localhost:5050*** (que es el puerto de pgadmin), seguido del ***email*** y ***contraseña*** especificadas en nuestro ***docker-composed.yml***. Al entrar se accederá al servidor con el usuario, contraseña y nombre de la base de datos especificados en este mismo, para luego crear una base de datos en la cual procederemos a crear las tablas para nuestro proyecto (con sus respectivos campos y relaciones) quedando de la siguiente manera:





• Registro e inicio de sesión funcionales.

Para nuestras secciones de ***registro*** y ***login*** funcionales, en nuestro archivo de ***main_routes.py*** para el ***login*** se establecieron los métodos ***GET*** y ***POST*** para mostrar y procesar los datos enviados en el formulario. Luego, si el método es igual a ***POST*** extrae lo que el usuario escribió en los campos de ***email*** y ***contraseña***, lo cual llama a una función ***authenticate_user*** (se verá más adelante) para verificar que los datos coincidan con un usuario previamente registrado, si es exitoso muestra un mensaje de “Inicio de sesión exitoso” y redirige a la página principal, en caso de que el email o contraseña no coincidan (o el usuario no esté previamente

registrado) mostrará un mensaje de “Correo o password incorrectos” y lo devolverá a la página de inicio.

```python
@main_bp.route('/login', methods=['GET','POST'])
def login():

    if request.method == 'POST':
        email = request.form['email']
        password = request.form['password']

        user = authenticate_user(email, password)

        if user:
            session['user_id'] = user.id
            session['user_email'] = user.email
            session['user_nombre'] = f"{user.nombre}"
            flash('Inicio de sesion exitoso', 'Success')
            return redirect(url_for('main_bp.inicio'))
        else:
            flash('Correo o password incorrectos','danger')
            return render_template('public/inicio.html', email='', password='')
```

Y para cerrar la sesión, con ***sesión.clear()*** se limpian los datos guardados durante la sesión, desconectando al usuario de esta y mostrando un mensaje de “Sesión cerrada correctamente” para redirigirlo a la página principal.

```python
@main_bp.route('/logout')
def logout():
    session.clear()
    flash('Sesión cerrada correctamente', 'success')
    return redirect(url_for('main_bp.inicio'))
```

Ahora, para nuestro registro creamos un modelo llamado ***user_model.py***, el cual defina la estructura de nuestra tabla ***usuarios*** en postgres. Primeramente se tiene el ***enum*** con los roles que pueden tener los usuarios (paciente, psicólogo y general), seguido de los campos de nuestra tabla (indicando la llave primaria, el tipo de dato, longitud de este, etc.)

```python
from app.models import db
from datetime import datetime
from sqlalchemy import Enum
import enum

class RoleEnum(enum.Enum):
    paciente = "paciente"
    psicologo = "psicologo"
    general = "general"

class User(db.Model):
    __tablename__ = 'usuarios'
    id = db.Column(db.Integer, primary_key=True)
    paterno = db.Column(db.String(15), nullable=False)
    materno = db.Column(db.String(15))
    nombre = db.Column(db.String(30), nullable=False)
    fecha_nacimiento = db.Column(db.Date, nullable=True)
    sexo = db.Column(db.String(1), nullable=True)
    direccion = db.Column(db.String(60), nullable=True)
    celular = db.Column(db.String(10), nullable=True)
    email = db.Column(db.String(254), unique=True, nullable=False)
    password = db.Column(db.String(128), nullable=False)
    role = db.Column(db.Enum(RoleEnum), default=RoleEnum.paciente, nullable=False)
    created_at = db.Column(db.DateTime, default=datetime.utcnow)

    def __repr__(self):
        return f"<User {self.email}>"
```

Luego en nuestro archivo ***main_routes.py*** tenemos nuestra parte para el registro de usuarios, donde también se implementaron los métodos de ***GET*** y ***POST***, si el método es igual a ***POST***, extrae los datos que el usuario ingresó en el formulario (apellidos paterno y materno, nombre, edad, etc.) de acuerdo a los campos de la base de datos en la tabla ***usuarios***.

```python
@main_bp.route('/registro', methods=['GET', 'POST'])
def registro():
    from app.models.user_model import RoleEnum

    if request.method == 'POST':
        email = request.form['email']

        try:
            # Captura de datos del formulario
            paterno = request.form['paterno']
            materno = request.form.get('materno')  # opcional
            nombre = request.form['nombre']
            edad = request.form.get('edad')
            fecha_nacimiento = request.form.get('fecha_nacimiento')
            sexo = request.form.get('sexo')
            direccion = request.form.get('direccion')
            celular = request.form.get('celular')
            email = request.form['email']
            password = request.form['password']
            role=RoleEnum.paciente
```

Lo siguiente es llamar al servicio (se mostrará más adelante) para registrar al usuario, asignando los datos capturados del formulario y pasarlos como argumentos con nombre.

```python
            # Llamar al service
            user, errores = register_user(
                paterno=paterno,
                materno=materno,
                nombre=nombre,
                fecha_nacimiento=fecha_nacimiento,
                sexo=sexo,
                direccion=direccion,
                celular=celular,
                email=email,
                password=password
            )
```

Y después se hacen excepciones para indicar si los campos de email o celular ya están registrados (ya que deben ser únicos) y renderiza el formulario de registro. Si

no están repetidos muestra un mensaje de “Registro exitoso. Ahora puedes iniciar sesión”, renderizando la página de inicio donde se encuentra la opción de ***login.***

```
91                if not user:
92                    for e in errores:
93                        if e == "email":
94                            flash("El correo ya está registrado.", "warning")
95                        elif e == "celular":
96                            flash("El número de celular ya está registrado.", "warning")
97                    return render_template("public/registro.html")
98                flash('Registro exitoso. Ahora puedes iniciar sesión.', 'success')
99                return redirect(url_for('main_bp.inicio'))
```

Entonces, ¿Dónde se crean las funciones de ***authenticate_user*** y ***register_user***? Para ellas creamos un archivo llamado ***auth_service.py***, donde creamos los dos métodos. Para ***authenticate_user*** recibe los parámetros de ***email*** y ***password*** y por parte de SLQALCHEMY nos brinda ***User.query.filter_by*** para buscar registros en la base de datos usando condiciones específicas (que en este caso busca el email y password).

```
1     # services/auth_service.py
2     from app.models.user_model import User
3     from app.models import db
4
5     def authenticate_user(email, password):
6         """
7         Busca un usuario que coincida con el email y password.
8         Retorna el objeto User si existe, o None si no.
9         """
10        user = User.query.filter_by(email=email, password=password).first()
11        return user
```

Y para el registro tenemos ***register_user***, el cual recibe los datos del formulario y se asegura que el email y celular no estén repetidos, o dará un error.

```python
def register_user(paterno, materno, nombre, fecha_nacimiento, sexo,
                  direccion, celular, email, password):
    """
    Registra un usuario nuevo.
    Retorna:
        - User: si se creó correctamente
        - None: si el email ya estaba registrado
    """
    errores = []

    if User.query.filter_by(email=email).first():
        errores.append("email")

    if celular and User.query.filter_by(celular=celular).first():
        errores.append("celular")

    if errores:
        return None, errores
```

Algo importante es la sesión de la base de datos, con ***db.session.add(user)*** prepara al usuario para ser guardado, agregándolo a la sesión de SLQALCHEMY, y con ***db.session.commit()*** ya se guarda al usuario, devolviendo a este último e indicando que no hubo errores.

```python
        # Guardar en la base de datos
        db.session.add(user)
        db.session.commit()
        return user, None
```

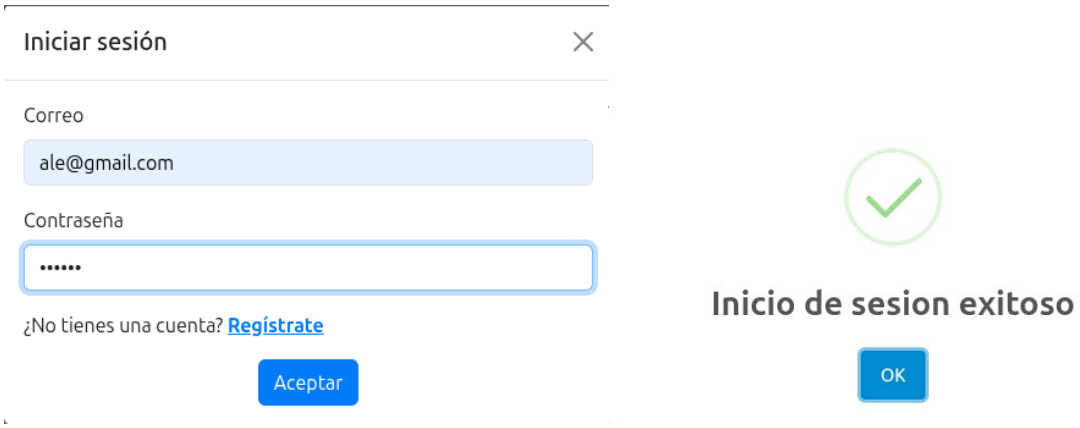
Y así se mira nuestra aplicación funcionando. Registramos a un paciente:

Y queda registrado en la base de datos:

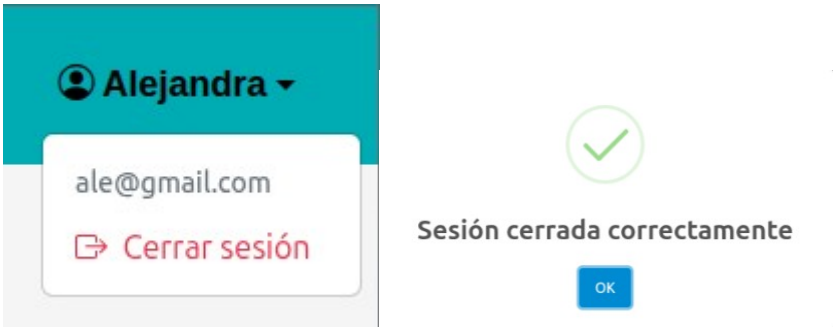
8	18	Rodriguez	Herrera	Maria Fernanda	2003-02-06	F	Pedregal
9	19	Medina	Espinoza	Alejandra	2007-06-20	F	Ejido Cuernavaca

6861234567	fer@gmail.com	fer123	paciente	2025-10-25 02:16:56.444546
6866641234	ale@gmail.com	ale123	paciente	2025-10-25 04:52:26.481563

Y cuando quiere iniciar sesión, lo hace de manera exitosa.



Y cuando inicia sesión, en el menú se muestra el nombre del paciente, así como su correo y la opción de cerrar sesión mediante un menú desplegable.



La observación de este sprint fue de nuevo la base de datos creada, ya que contiene relaciones que no estaban previstas en el diseño, volviendo a generar problemas de redundancia. Se arreglará y presentará para el tercer sprint que es el día 7 de noviembre del 2025.

Sprint 3: 27 oct → 7 nov

Entregables:

• Función de agendar cita (Sin correo de recordatorio).

• Estados de cita: Pendiente, Aceptada (Si es el mismo día).

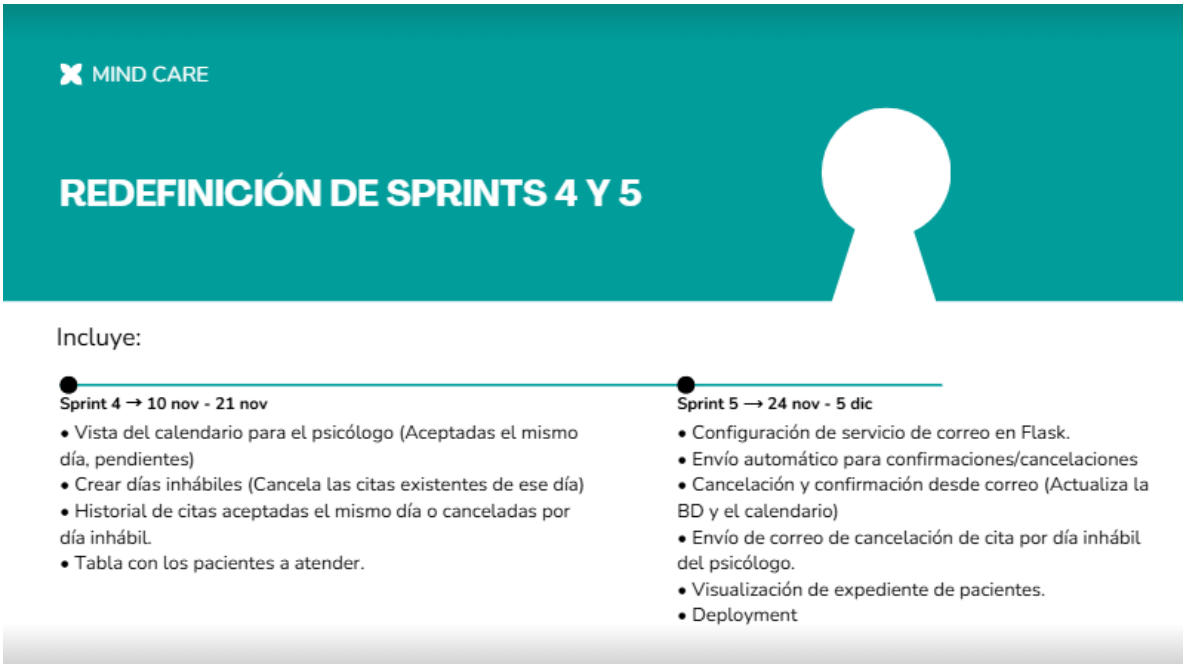
• Historial de citas.

• Vista de Calendario para el usuario.

Notas:

• Redefinición de Sprints.

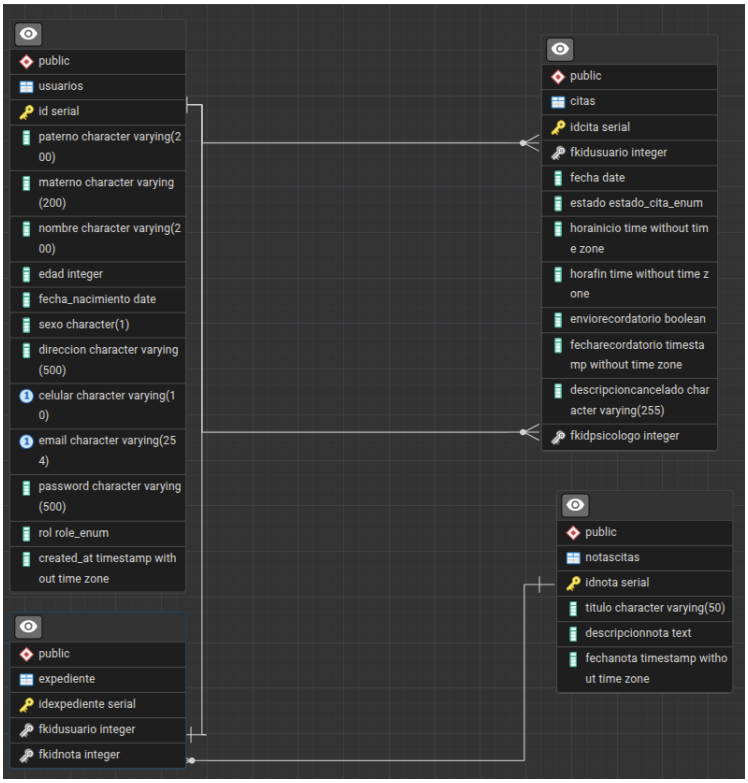
Como se unió un nuevo integrante al equipo, se nos pidió redefinir nuestros sprints, los cuales quedaron de la siguiente manera:



Quedando este Sprint 3 de la misma manera y cambiando los sprint 4 y 5, los cuales son de mayor peso. En un principio se tenia previsto que la tarea de la tabla con los pacientes a atender se llevaría a cabo hasta el sprint 5, pero se recorrió para el sprint 4, esto para minimizar un poco la carga.

• Corrección de diagrama de base de datos

En el sprint 2 se nos hizo el comentario del diagrama de relación de nuestra base de datos, el cual tenía una relación que no iba. Ya corrigiéndolo quedó de la siguiente manera, con dos relaciones de la llave foránea del usuario hacía las citas.



• Validaciones para el formulario de registro

Así mismo, se hicieron validaciones para lo que es el formulario de registro de pacientes en nuestro archivo ***user_validator.py*****.** De primera instancia se tiene especificando que los campos de nombre, apellidos, correo, contraseña y celular sean obligatorios, en caso de que se ingresen se obtienen con el ***get***, en caso contrario muestra un mensaje de error que los campos son obligatorios.

Ahora, para validar el nombre y los apellidos (así como los demás campos) se utilizaron expresiones regulares. Donde, se valida que solo se ingresen letras, si intentan ingresar números o algo más que no sean letras, muestra un mensaje de que los campos solo pueden contener letras:

```
# Campos obligatorios
obligatorios = ["nombre", "paterno", "email", "celular", "password"]
for campo in obligatorios:
    if not data.get(campo):
        errores.append(f"El campo '{campo}' es obligatorio.")

# Validar nombre y apellidos
patron_texto = r"^[A-Za-zÁÉÍÓÚáéíóúñÑ\s]+$"
for campo in ["nombre", "paterno", "materno"]:
    if data.get(campo) and not re.match(patron_texto, data[campo]):
        errores.append(f"El campo '{campo}' solo puede contener letras.")
```

De igual manera, se tienen para los campos de email y celular, validando que se ingrese la estructura correcta para cada uno. En caso contrario mostrará un mensaje diciendo que no es válido:

```
# Validar email
email = data.get("email", "")
patron_email = r'^[\w\.-]+@[\w\.-]+\.\w+$'
if email and not re.match(patron_email, email):
    errores.append("El correo electrónico no es válido.")

# Validar celular
celular = data.get("celular", "")
if celular and (not re.match(r'^[0-9]{10}$', celular)):
    errores.append("El número celular debe tener 10 dígitos.")
```

Y de igual manera para los campos de contraseña y fecha de nacimiento:

```
# Validar contraseña
password = data.get("password", "")
if password and (len(password) < 8 or len(password) > 20):
    errores.append("La contraseña debe tener entre 8 y 20 caracteres.")

# Validar fecha de nacimiento (si se envía)
fecha_nacimiento = data.get("fecha_nacimiento")
if fecha_nacimiento:
    try:
        datetime.strptime(fecha_nacimiento, "%Y-%m-%d")
    except ValueError:
        errores.append("La fecha de nacimiento no tiene un formato válido (YYYY-MM-DD).")

return errores
```

Ya por último, se tiene para validar que no haya emails o números de celular registrados:

```python
@staticmethod
def check_duplicates(data):
    """
    Valida que email y celular no existan en la base de datos.
    Retorna lista de errores.
    """
    errores = []
    if "email" in data and User.query.filter_by(email=data["email"]).first():
        errores.append("El correo ya está registrado.")

    if "celular" in data and data["celular"] and User.query.filter_by(celular=data["celular"]).first():
        errores.append("El número celular ya está registrado.")

    return errores
```

Ahora, comenzando con lo planeado para entregar en este sprint, se hizo el calendario para que los pacientes puedan agendar citas. Este mismo se realizó con FullCalendar js, el cual es una herramienta de Javascript para manejar calendarios dinámicos mediante código HTML, importando primeramente los scripts necesarios para que este funcione (que sería el primero), y los otros dos son para la funcionalidad de agendar y para mostrar los popups de alerta. De igual manera se le importan los estilos para nuestra página con un css aparte:

```html
<script src="https://cdn.jsdelivr.net/npm/fullcalendar@6.1.8/index.global.min.js"></script>
<script src="{{ url_for('static', filename='js/paciente/agendar.js') }}"></script>
<script src="https://cdn.jsdelivr.net/npm/sweetalert2@11"></script>
```

```html
{% block style %}
  <link rel="stylesheet" href="{{ url_for('static', filename='css/paciente/style_inicio_paciente.css') }}">
{% endblock%}
```

Aquí se está definiendo la opción de ***Regresar*** para regresar a la página de inicio (***user_bp.inicio***), en dado caso de que el paciente quiera hacerlo. Así mismo se muestra un mensaje de “¿Que día te gustaría agendar?”, seguido de leyendas con los colores para indicar los estados de las citas/cupos para agendar, muestra el día actual, si está ocupado, si un día esta inhábil, etc. Y por último el ***div*** que es donde está el calendario:

```
<div class="back">
  <a href="{{ url_for('user_bp.inicio') }}">
    <img src="{{ url_for('static', filename='img/back.png') }}">
    <span>Regresar</span>
  </a>
  <h1>¿Que día te gustaría agendar?</h1>
  <div class="leyendas">
    <div class="leyenda-item disponible">
      <span class="cuadro"></span> Lugar disponible
    </div>
    <div class="leyenda-item ocupado">
      <span class="cuadro"></span> Lugar ocupado
    </div>
    <div class="leyenda-item inhabil">
      <span class="cuadro"></span> Día inhabil
    </div>
    <div class="leyenda-item hoy">
      <span class="cuadro"></span> Día de noy
    </div>
  </div>
</div>

<div id='calendar'></div>
```

Pasando al archivo ***agendar.js***, tenemos una variable para guardar nuestro calendario (esto para manipularlo), así como el id del usuario para manejar las sesiones, ósea, que el paciente pueda manejar su sesión. Luego se obtiene la fecha actual y la convierte en formato día, mes, año. Seguido se manda un endpoint para listar las citas del día actual y convertirlas en formato Json:

```
document.addEventListener('DOMContentLoaded', async function () {
  const calendarEl = document.getElementById('calendar');

  // ID del usuario logueado (inyectado desde backend, por ejemplo en un script)
  const userId = parseInt(document.getElementById('user_id').value, 10);

  // Fecha actual
  const hoy = new Date();
  const hoyISO = hoy.toISOString().split('T')[0];

  // Cargar citas desde el backend
  const citasResponse = await fetch('/api/citas/listar');
  const citas = await citasResponse.json();
```

Luego se valida que las citas que no sean del usuario logeado se muestren como ocupadas (ya que no le importa si otras citas están pendientes o ya fueron aceptadas). En cambio, si las citas son del usuario que esta actualmente logeado, se muestran los estados de estas, ya que han sido las que él ha agendado:

```
function getClaseCita(cita) {
// Si no es del usuario logeado, mostrar como ocupada (azul)
if (cita.fkidusuario !== userId) {
  return 'cita-ocupada';
}

else if(cita.fkidusuario === userId){
  // Si es del usuario logeado, mostrar según estado
  switch (cita.estado) {
    case 'pendiente':
      return 'cita-pendiente';
    case 'aceptada':
      return 'cita-aceptada';
    case 'cancelada':
      return 'cita-cancelada';
    default:
      return '';
  }
```

Lo siguiente es convertir los datos crudos en eventos para el calendario, lo que quiere decir que se hace un mapeo de las citas haciendo una comparación del id del paciente (para mostrar sus citas). Y una vez comprobado devuelve las citas del usuario, mostrando el estado de estas (pendiente, aceptada, cancelada) así como la hora de inicio y fin, en caso contrario (citas ajenas) solo muestra el estado en “ocupado”.

```
// Convertir citas en eventos para el calendario
const eventos = citas.map(c => {
  const esDelUsuario = c.fkidusuario === userId;

  return {
    title: esDelUsuario
      ? c.estado.charAt(0).toUpperCase() + c.estado.slice(1)  // Pendiente / Aceptada / Cancelada
      : 'Ocupado', // 👈 Para otros usuarios, siempre muestra “Ocupado”
    start: `${c.fecha}T${c.horainicio}`,
    end: `${c.fecha}T${c.horafin}`,
    classNames: [getClaseCita(c)],
    editable: false
  };
});
```

Ahora, se da inicio a lo que es nuestro calendario (de acuerdo al div calendar de nuestro html). Aquí se le esta dando una vista para que se muestre de manera semanal (lunes-viernes), mostrando los días en español con formato de nombre del día y número de este, así mismo se define la hora de inicio y fin de acuerdo al horario del consultorio con una duración de una hora (duración de las sesiones), y de igual manera se le da formato de 24 horas. Luego se define que tenga las funciones para desplazarse de semanas previas, siguientes y regresarse al día actual. Por último con validRange se delimita el agendado de citas, ya que no permite agendar en días pasados así como horas pasadas a la actual:

```
// Inicializar el calendario
const calendar = new FullCalendar.Calendar(calendarEl, {
  initialView: 'timeGridWeek',
  locale: 'es',
  dayHeaderFormat: { weekday: 'long', day: 'numeric' },
  allDaySlot: false,
  slotMinTime: '10:00:00',
  slotMaxTime: '21:00:00',
  slotDuration: '01:00:00',
  slotLabelFormat: { hour: 'numeric', hour12: true },
  headerToolbar: {
    left: 'prev',
    center: 'title',
    right: 'today next'
  },
  buttonText: { today: 'Hoy' },
  events: eventos,

  validRange: { start: hoyISO }, // No permite días pasados
```

Aquí se definen los días de sábado y domingo como inhábiles (0 y 6), y los marca en un color gris. Por otro lado, para marcar el día actual lo hace con otro color:

```
dayCellDidMount: function (info) {
  const day = info.date.getDay();
  const isToday = info.date.toDateString() === hoy.toDateString();

  if (day === 0 || day === 6) {
    info.el.style.backgroundColor = '#bdc3c7'; // inhábil
  }
  if (isToday) {
    info.el.style.backgroundColor = '#A4FDE7'; // color del día actual
  }
},
```

Entonces, cuando un paciente quiera agendar y de click en una de las celdas del calendario se mostrará un popup con la hora seleccionada, así como otros datos que se dirán más adelante:

```
dateClick: async function (info) {
  const fecha = info.dateStr.split('T')[0];
  const horaInicio = info.date.toLocaleTimeString('en-GB', {
    hour: '2-digit', minute: '2-digit', second: '2-digit'
  });
  const horaFinObj = new Date(info.date.getTime() + 60 * 60 * 1000);
  const horaFin = horaFinObj.toLocaleTimeString('en-GB', {
    hour: '2-digit', minute: '2-digit', second: '2-digit'
  });
```

Seguido de esto es más que nada validar que el usuario no pueda agendar en días inhábiles o pasados, al igual que en horas pasadas:

```
// Validaciones
const dia = info.date.getDay();
if (dia === 0 || dia === 6) {
  Swal.fire('Día inhábil', 'No puedes agendar en fines de semana.', 'warning');
  return;
}

const fechaClick = new Date(fecha);
if (fechaClick < new Date(hoyISO)) {
  Swal.fire('Fecha inválida', 'No puedes agendar en días pasados.', 'error');
  return;
}

const ahora = new Date();
if (fecha === hoyISO && info.date < ahora) {
  Swal.fire('Hora inválida', 'No puedes agendar en una hora pasada de hoy.', 'error');
  return;
}
```

Lo siguiente a mostrar para agendar la cita (en el popup) es el nombre del psicólogo, para lo cual se manda un endpoint para consultarlo:

```
// Obtener psicólogo real
const psicologoResponse = await fetch('/api/psicologo');
const psicologo = await psicologoResponse.json();
```

Ahora aquí se construye el popup con los datos de la hora de inicio y fin de la sesión y el nombre del psicólogo, donde si el paciente la confirmase guardan en una variable ***data*:**

```
// Confirmar cita
const result = await Swal.fire({
  title: '¿Agendar cita?',
  html: `
    <p><b>Fecha:</b> ${fecha}</p>
    <p><b>Hora:</b> ${horaInicio} - ${horaFin}</p>
    <p><b>Psicólogo:</b> ${psicologo.nombre} ${psicologo.apellidopaterno}</p>
  `,
  icon: 'question',
  showCancelButton: true,
  confirmButtonText: 'Sí, agendar',
  cancelButtonText: 'Cancelar'
});

if (result.isConfirmed) {
  const data = {
    fecha,
    horainicio: horaInicio,
    horafin: horaFin
  };
```

Algo que manejamos por lógica es que si el paciente agenda el mismo día automáticamente el estado de la cita es “aceptada”:

```
// Si la cita es hoy, marcar como aceptada
if (fecha === hoyISO) data.estado = 'aceptada';

const resp = await fetch('/api/citas', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify(data)
});
```

Y ya que el paciente ha agendado su cita se muestra en el calendario, mostrando los datos antes mencionados en la hora elegida:

```
      const json = await resp.json();

      if (resp.ok) {
        Swal.fire('Éxito', 'Cita agendada correctamente', 'success');
        calendar.addEvent({
          title: data.estado === 'aceptada' ? 'Ocupado' : 'Pendiente',
          start: `${fecha}T${horaInicio}`,
          end: `${fecha}T${horaFin}`,
          classNames: [data.estado === 'aceptada' ? 'cita-aceptada' : 'cita-pendiente'],
          editable: false
        });
      } else {
        Swal.fire('Error', json.error || 'No se pudo agendar la cita', 'error');
      }
    }
  }
});

calendar.render();
```

Una vez el paciente haya agendado su cita, esta irá a su historial. Aquí se extrae el id del paciente para identificar sus citas y mostrarlas, en dado caso que no se encuentre el id se mostrará un mensaje de error, pero si sí lo encuentra se manda un endpoint para traerlo y lo convierte a Json, y después se empieza con la construcción de la tabla para mostrar las citas, pero en caso de que no haya se mostrará un mensaje de que no hay citas registradas:

```
document.addEventListener("DOMContentLoaded", async function() {
  const urlParams = new URLSearchParams(window.location.search);
  const userId = urlParams.get("user_id");

  if (!userId) {
    console.error("No se encontró el ID del usuario en la URL");
    return;
  }

  try {
    const response = await fetch(`/api/historial/${userId}`);
    const data = await response.json();

    const tbody = document.querySelector("#tablaHistorial tbody");
    tbody.innerHTML = "";

    if (data.length === 0) {
      tbody.innerHTML = `
        <tr><td colspan="7">No tienes citas registradas.</td></tr>
      `;
      return;
    }
```

Entonces, si el paciente tiene citas agendadas traerá desde el back los datos correspondientes de esta (idcita, fecha, horainicio, horafin, etc.) y con ellos creará las filas de la tabla del historial:

```
    data.forEach(cita => {
      const row = document.createElement("tr");

      row.innerHTML = `
        <td>${cita.idcita}</td>
        <td>${cita.fecha}</td>
        <td>${cita.horainicio}</td>
        <td>${cita.horafin}</td>
        <td>${cita.estado}</td>
        <td>${cita.descripcioncancelado || "—"}</td>
        <td>${cita.psicologo_nombre}</td>
      `;

      tbody.appendChild(row);
    });
  } catch (error) {
    console.error("Error al obtener el historial:", error);
  }
});
```

Y esta es la estructura html para la tabla, donde como se puede ver se obtienen los datos con ***cita.fecha, cita.horainicio,*** etc:

```
<div class="tabla-citas">
  <table>
    <thead>
      <tr>
        <th>Fecha</th>
        <th>Hora Inicio</th>
        <th>Hora Fin</th>
        <th>Psicólogo</th>
        <th>Estado</th>
      </tr>
    </thead>
    <tbody>
      {% for cita in citas %}
      <tr>
        <td>{{ cita.fecha }}</td>
        <td>{{ cita.horainicio }}</td>
        <td>{{ cita.horafin }}</td>
        <td>{{ cita.psicologo_nombre if cita.psicologo_nombre else 'No asignado' }}</td>
        <td>{{ cita.estado|capitalize }}</td>
      </tr>
      {% else %}
      <tr>
        <td colspan="5">No hay citas registradas</td>
      </tr>
      {% endfor %}
    </tbody>
  </table>
```

¿Y cómo se obtienen los datos del psicólogo? Esto se hace por medio de interfaces e implementaciones, donde en la interfaz se declara una clase que hereda ABD, la cual permite definir una interfaz que otras clases deben implementar, a la vez que declara un método para que sea implementado por las subclases:

```python
class PsicologoService(ABC):

    @abstractmethod
    def obtener_psicologo(self):
        """Devuelve el psicólogo único registrado en el sistema"""
        pass
```

Y en la implementación se implementa el método de la interfaz, aquí busca en la base de datos al psicólogo y devuelve sus datos:

```python
class PsicologoServiceImpl(PsicologoService):

    def obtener_psicologo(self):
        """Devuelve el primer psicólogo disponible (por simplicidad)"""
        try:
            psicologo = User.query.filter_by(rol='psicologo').first()
            if not psicologo:
                return {"error": "No hay psicólogos registrados"}

            return {
                "idusuario": psicologo.id,
                "nombre": psicologo.nombre,
                "apellidopaterno": psicologo.paterno,
                "apellidomaterno": psicologo.materno,
                "correo": psicologo.email
            }
        except Exception as e:
            return {"error": f"Error al obtener psicólogo: {str(e)}"}
```

Y para las citas se tiene lo mismo, primero en la interfaz se hacen los métodos para agendar y obtener dichas citas:

```python
class CitaService(ABC):

    @abstractmethod
    def agendar_cita(self,data):
        pass

    @abstractmethod
    def obtener_citas(self):
        pass
```

Y en la implementación se tiene primero el método para agendar la cita, donde se obtienen el psicólogo, fecha actual y fecha de la cita, agregando la condición de que si la cita es agendada el mismo día automáticamente el estado es aceptada, en caso contrario la marca como pendiente. Luego, crea una instancia de la cita con sus respectivos datos, y por último confirma los cambios:

```
class CitaServiceImpl(CitaService):

    def agendar_cita(self, data):
        try:
            psicologo = User.query.filter_by(rol='psicologo').first()
            hoy = date.today()
            fecha_cita = datetime.strptime(data["fecha"], "%Y-%m-%d").date()

            # Si la cita es hoy, cambia estado a 'aceptada', si no, 'pendiente'
            estado_cita = "aceptada" if fecha_cita == hoy else "pendiente"

            if not psicologo:
                return {"error": "No hay psicólogo registrado en el sistema"}

            nueva_cita = Cita(
                fkidusuario=data["fkidusuario"],
                fkidpsicologo=psicologo.id,
                fecha=data["fecha"],
                horainicio=data["horainicio"],
                horafin=data["horafin"],
                estado=estado_cita,
                descripcioncancelado=data.get("descripcioncancelado")
            )
```

```
db.session.add(nueva_cita)
db.session.commit()
return {"message": "Cita creada correctamente", "idcita": nueva_cita.idcita}
```

De igual manera se tiene el método para obtener las citas, en el cual consulta las citas que día actual o después, y ya que obtiene un resultado las guarda e itera sobre cada una de ellas con sus datos y devuelve el resultado:

```
def obtener_citas(self):
    try:
        hoy = date.today()
        citas = Cita.query.filter(Cita.fecha >= hoy).all()

        resultado = []
        for cita in citas:
            resultado.append({
                "idcita": cita.idcita,
                "fkidusuario": cita.fkidusuario,
                "fkidpsicologo": cita.fkidpsicologo,
                "fecha": cita.fecha.strftime("%Y-%m-%d"),
                "horainicio": cita.horainicio.strftime("%H:%M"),
                "horafin": cita.horafin.strftime("%H:%M"),
                "estado": cita.estado
            })

        return resultado
```

Así mismo se tiene el método de obtener el historial, donde se hace una consulta a la tabla de citas para obtener los datos de esta:

```python
def obtener_historial(self, user_id):
    query = text("""
        SELECT
            c.idcita,
            c.fecha,
            c.horainicio,
            c.horafin,
            c.estado,
            c.descripcioncancelado,
            u.id AS usuario_id,
            CONCAT(u.nombre, ' ', u.paterno, ' ', COALESCE(u.materno, '')) AS usuario_nombre,
            p.id AS psicologo_id,
            CONCAT(p.nombre, ' ', p.paterno, ' ', COALESCE(p.materno, '')) AS psicologo_nombre
        FROM citas c
        JOIN usuarios u ON u.id = c.fkidusuario
        JOIN usuarios p ON p.id = c.fkidpsicologo
        WHERE u.id = :user_id
        ORDER BY c.fecha, c.horainicio
    """)

    result = db.session.execute(query, {"user_id": user_id})
    return [dict(row._mapping) for row in result]
```

Por la parte de los modelos, comenzamos por el del usuario, en el cual se especifican el ***enum*** para los roles, así como los campos de la tabla de la base de datos:

```python
class RoleEnum(enum.Enum):
    paciente = "paciente"
    psicologo = "psicologo"
    general = "general"

class User(db.Model):
    __tablename__ = 'usuarios'
    id = db.Column(db.Integer, primary_key=True)
    paterno = db.Column(db.String(15), nullable=False)
    materno = db.Column(db.String(15))
    nombre = db.Column(db.String(30), nullable=False)
    edad = db.Column(db.Integer, nullable=False)
    fecha_nacimiento = db.Column(db.Date, nullable=True)
    sexo = db.Column(db.String(1), nullable=True)
    direccion = db.Column(db.String(60), nullable=True)
    celular = db.Column(db.String(10), unique=True, nullable=False)
    email = db.Column(db.String(254), unique=True, nullable=False)
    password = db.Column(db.String(128), nullable=False)
    rol = db.Column(db.Enum(RoleEnum), default=RoleEnum.paciente, nullable=False)
    created_at = db.Column(db.DateTime, default=datetime.utcnow)
```

En el modelo para las citas se tiene algo parecido, a excepción de que el enum es de los estados de las citas:

```python
class EstadoCitaEnum(enum.Enum):
    aceptada = "aceptada"
    pendiente = "pendiente"
    cancelada = "cancelada"
    finalizada = "finalizada"

class Cita(db.Model):
    __tablename__ = 'citas'

    idcita = db.Column(db.Integer, primary_key=True)
    fkidusuario = db.Column(db.Integer, db.ForeignKey('usuarios.id'), nullable=False)
    fkidpsicologo = db.Column(db.Integer, db.ForeignKey('usuarios.id'), nullable=False)
    fecha = db.Column(db.Date, nullable=False)
    estado = db.Column(db.String(20), default='pendiente', nullable=False)
    horainicio = db.Column(db.Time, nullable=False)
    horafin = db.Column(db.Time, nullable=False)
    enviorecordatorio = db.Column(db.Boolean, default=False)
    fecharecordatorio = db.Column(db.DateTime)
    descripcioncancelado = db.Column(db.String(255))
```

Ya para finalizar, en el archivo de ***user_routes.py*** se agregó la ruta para el historial, validando el id del usuario:

```python
@user_bp.route('/historial')
@login_required
def historial():
    user_id = request.args.get("user_id", g.user.id)
    return render_template('paciente/historial.html', user=g.user)
```

Ahí mismo se definen las rutas para mostrar el historial mediante la obtención del id del usuario, donde en la primera se obtienen los datos del historial, y en la segunda se renderiza la página html para la vista:

```python
@user_bp.route('/api/historial/<int:user_id>', methods=['GET'])
@login_required
def obtener_historial(user_id):
    data = cita_service.obtener_historial(user_id)
    return jsonify(data)

@user_bp.route('/historial/<int:user_id>')
def historial_page(user_id):
    citas = cita_service.obtener_historial(user_id)
    return render_template('paciente/historial.html', citas=citas)
```

Ahora en el archivo de ***psicologo_routes.py*** se obtienen los datos del psicólogo en formato Json, así como una lista de psicólogos (en caso de que haya más):

```python
@psicologo_bp.route('/api/psicologo', methods=['GET'])
def obtener_psicologo():
    """Devuelve el primer psicólogo disponible"""
    data = psicologo_service.obtener_psicologo()
    status = 200 if "error" not in data else 404
    return jsonify(data), status

@psicologo_bp.route('/api/psicologos', methods=['GET'])
def listar_psicologos():
    """Devuelve la lista completa de psicólogos"""
    data = psicologo_service.listar_psicologos()
    status = 200 if "error" not in data else 400
    return jsonify(data), status
```

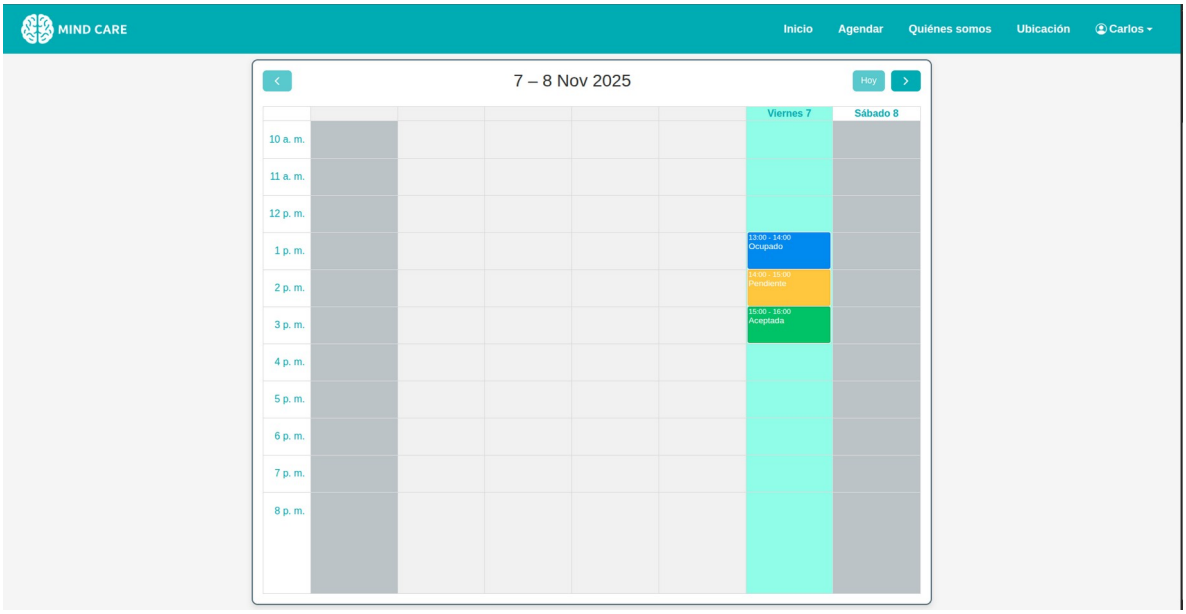
Y por último se tiene el archivo ***citas_routes.py***, en donde se tiene una ruta para listar las citas por medio de un método ***get*** (para obtenerlas) y las devuelve para mostrarlas en el calendario:

```python
@citas_bp.route('/api/citas/listar', methods=['GET'])
@login_required
def listar_citas():
    """Devuelve todas las citas registradas (para mostrar en el calendario)"""
    citas = cita_service.obtener_citas()  # función que debes tener en tu service
    return jsonify(citas), 200
```

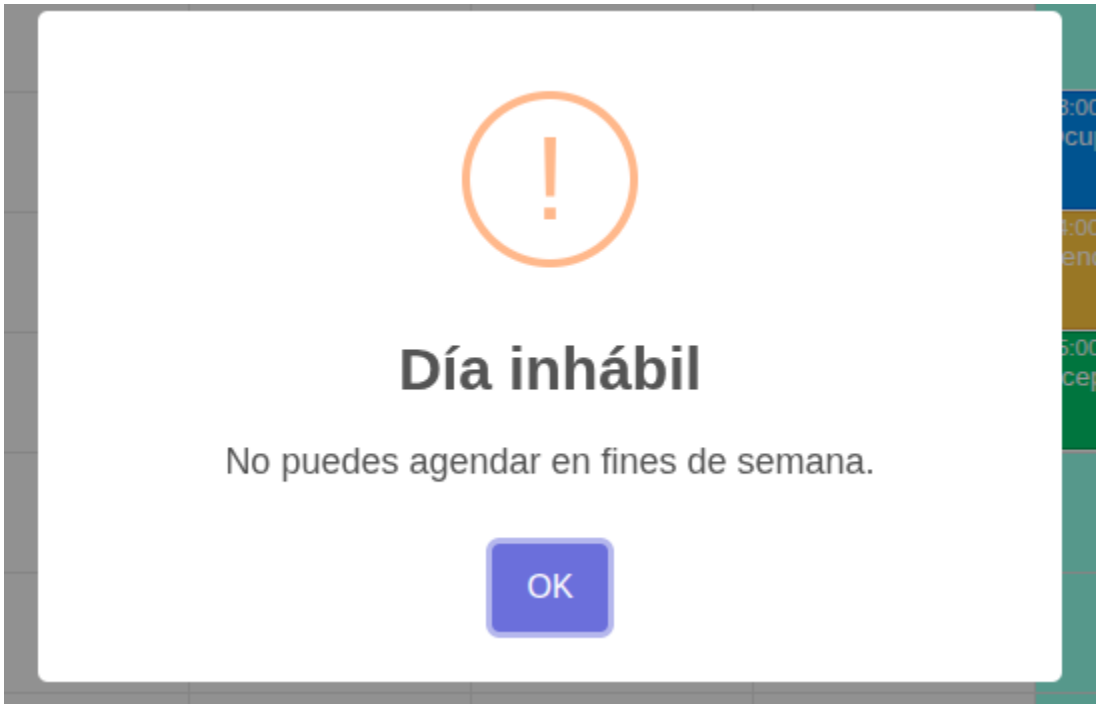
De igual manera de tiene otra ruta para crear citas con el método ***post***, validando que tenga los campos mínimos (fecha, hora de inicio y fin) y relacionándola con el id del paciente:

```python
@citas_bp.route('/api/citas', methods=['POST'])
@login_required
def agendar_cita():
    """Crea una nueva cita para el usuario logueado"""
    data = request.get_json()

    # Validar datos mínimos
    if not all(k in data for k in ("fecha", "horainicio", "horafin")):
        return jsonify({"error": "Faltan campos obligatorios"}), 400

    data["fkidusuario"] = g.user.id
    result = cita_service.agendar_cita(data)
    status = 200 if "error" not in result else 400
    return jsonify(result), status
```

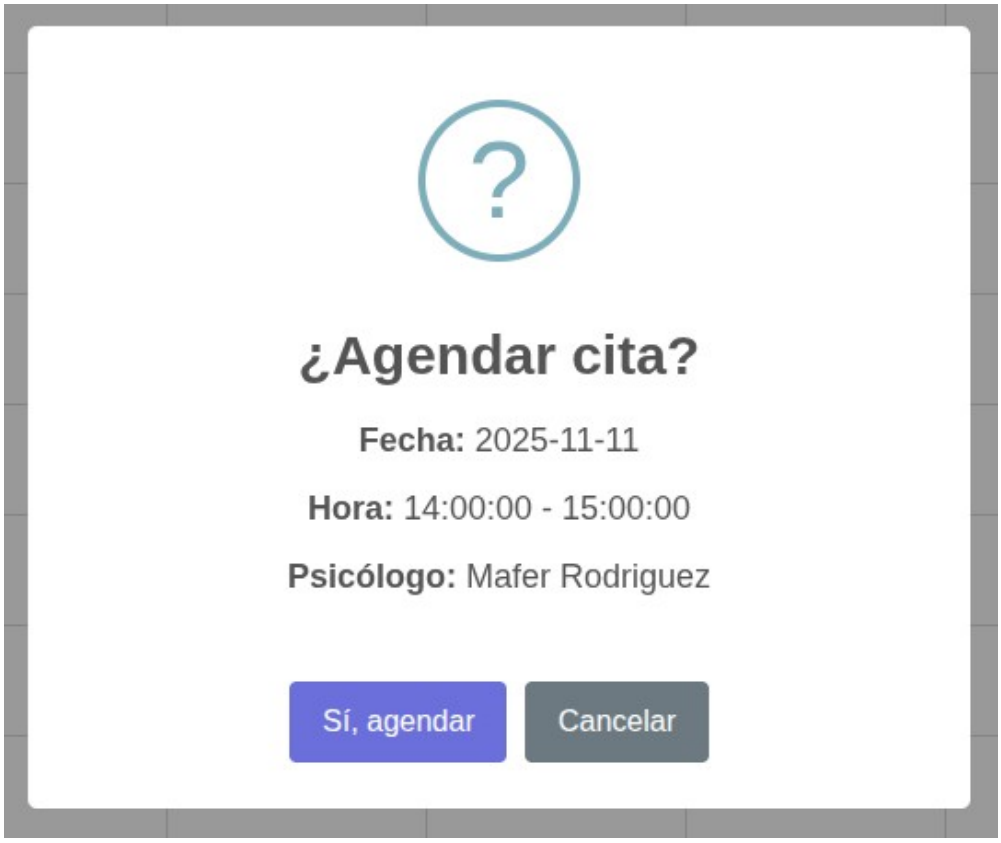
Imagenes.



Dia inhabil.



Agendar cita.



Historial.

Agendar sin iniciar sesion.