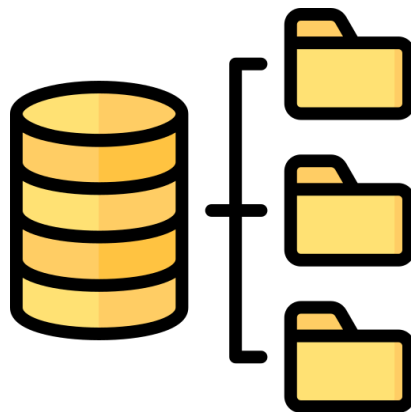
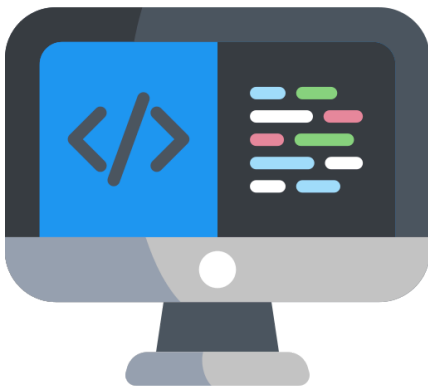


Proyecto Final

Módulo

Programador



ALUMNOS:

- | | |
|-------------------|----------------------|
| ■ Luciano Lujan | ■ Vittorio Durigutti |
| ■ Joaquín Zalazar | ■ Lisandro Juncos |

Plazo máximo de entrega: 10 de junio de 2024.

Objetivo:

Desarrollar una pequeña aplicación que demuestre la integración de conceptos de programación y bases de datos, y que aborde aspectos éticos y profesionales según la normativa vigente en la provincia de Córdoba.

Pautas de grupo y entregable:

El presente trabajo práctico final integrador, se plantea para realizarse en grupo de hasta 4 alumnos. Una vez confeccionado se debe entregar un archivo en PDF que contenga lo siguiente:

1. Código fuente en Python que incluya la temática seleccionada a desarrollar, con las respectivas consultas a la base de datos.
2. Modelo relacional normalizado en 3FN.
3. Consultas SQL.
4. Informe de aspectos éticos y profesionales.

Se deberá entregar una ficha informativa que incluya:

- Título del proyecto.
- Nombres completos de los integrantes del grupo.
- Fecha de entrega.
- Relato de problemática planteada (máximo 200 palabras).

Extensión y formato:

El informe final deberá tener una extensión de entre 10 y 20 páginas, escritas en letra Arial 12, a doble espacio, con márgenes de 2,5 cm. El proyecto deberá presentarse en formato digital (PDF).

Cada grupo deberá presentar su proyecto ante los docentes del módulo. La defensa oral tendrá una duración de 15 minutos por grupo y se evaluará la capacidad de los integrantes para explicar y defender su trabajo de manera clara y concisa. Los integrantes del grupo deberán exponer de manera conjunta, distribuyendo el tiempo de manera equitativa.

Se evaluará la presentación y funcionamiento del proyecto, la capacidad de respuesta a las preguntas y la defensa general del trabajo.

Problemática Planteada:

El Dr. Javier desea implementar un sistema de gestión de citas médicas en su clínica para optimizar el proceso de atención a los pacientes. El sistema permitirá registrar y gestionar la información de los pacientes, médicos, y las citas de manera eficiente y organizada. El sistema permitirá a los administradores de la clínica programar y gestionar citas, registrando información clave como la fecha, hora, especialidad y motivo de la consulta, así como los datos del paciente y del médico asignado.

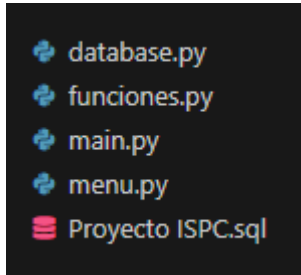
Además, los médicos podrán acceder a la información de contacto y direcciones de los pacientes para facilitar la comunicación y el seguimiento.

Este programa mejorará la eficiencia operativa de la clínica al reducir el tiempo y esfuerzo necesarios para gestionar las citas y la información de los pacientes, permitiendo un mejor servicio y atención médica.

Trabajo Final Integrador

Código fuente Python

Estructura del proyecto



En el archivo `database.py` encontramos solo la función para ingresar el “**user**” y “**password**”, y los demás datos relevantes para establecer conexión con el servidor personal de **MySQL** para realizar la conexión, ya que los métodos para **obtener**, **insertar**, **actualizar** y **eliminar** datos se encuentran dentro del código principal.



Se colocó el host, puerto y nombre de la base de datos como valores fijos, a fin de optimizar el tiempo cuando se ejecuta el programa.

```
import mysql.connector

def conectar_db():
    host = '127.0.0.1'
    port = '3306'
    user = input("Ingrese el nombre de usuario: ")
    password = input("Ingrese la contraseña: ")
    database = 'PROYECTO_ISPC'

    conn = mysql.connector.connect(
        host=host,
        port=port,
        user=user,
        password=password,
        database=database
    )
    cursor = conn.cursor()
    return conn, cursor
```

conn utiliza la función connect del módulo mysql.connector para establecer la conexión con la base de datos, y siguiente se crea un objeto cursor , usando el método cursor de la conexión. Terminan devolviendo conn y cursor, y es lo que utilizaremos luego para ejecutar las consultas SQL en las funciones

En el archivo  **funciones.py** encontramos, vaya la redundancia, todas las **funciones** para **interactuar** con la base de datos  **PROYECTO_ISPC**. Algunas funciones son:

Añadir un nuevo paciente y médico, respectivamente, a la base de datos:

```
# Funcion para añadir un nuevo paciente a la base de datos
def agregar_paciente(db, cursor, Nombre, Apellido, DNI, Fecha_Nacimiento, Telefono1, Telefono2, Email, Provincia, Ciudad, Calle, CP, Numero, Depto):
    Fecha_Nacimiento = datetime.strptime(Fecha_Nacimiento, '%d-%m-%Y').strftime('%Y-%m-%d')
    try:
        cursor.execute("INSERT INTO pacientes (DNI, Nombre, Apellido, Fecha_Nacimiento, Telefono1, Telefono2, Email, Provincia, Ciudad, Calle, CP, Numero, Depto) VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)", (DNI,
        Nombre,
        Apellido,
        Fecha_Nacimiento,
        Telefono1,
        Telefono2,
        Email,
        Provincia,
        Ciudad,
        Calle,
        CP,
        Numero,
        Depto))

        db.commit()
        print("Paciente ahadido con exito!")
    except mysql.connector.errors.IntegrityError:
        print("Ya existe un paciente registrado con los mismos datos.")

# Funcion para añadir un nuevo medico a la base de datos
def agregar_medico(db, cursor, Nombre, Apellido, DNI, Especialidad, Email):
    try:
        cursor.execute("INSERT INTO medicos (Nombre, Apellido, DNI, Especialidad, Email) VALUES (%s, %s, %s, %s, %s)", (Nombre, Apellido, DNI, Especialidad, Email))
        db.commit()
        print("Nuevo miembro del personal ahadido con exito!")
    except mysql.connector.errors.IntegrityError:
        print("Ya existe un registro de personal bajo los mismos datos.")
```

Todas las funciones tienen una estructura similar. Nombramos la función, y se indican los parámetros requeridos. Además de los parámetros requeridos para la carga, cada función solicita “db” y “cursor” siendo estos los parámetros de entrada que permiten ejecutar los comandos SQL. Se compone por los bloques “try” y “except” ante la presencia de algún error al momento de ingresar los valores de los atributos. “cursor.execute” que ejecuta el comando SQL para la carga de los datos a la tabla indicada. Y el “db.commit” necesario para asegurar que los cambios en la base de datos sean permanentes

En la función “agregar_paciente” encontramos además previo al “try”, el uso de la función strptime, que corresponde al módulo datetime, para convertir el valor de fecha de nacimiento que se ingresa en formato string, a un objeto datetime. Seguido de .strftime('%Y-%m-%d') que convierte el formato, al que utiliza/reconoce MySQL.

Crear un turno:

```
# Funcion para crear un turno. Es extensa por la cantidad de variables involucradas
def crear_turno(conn, cursor, id_paciente, id_medico, Fecha, Hora, Especialidad, Motivo_consulta):

    #Misma linea de codigo que en la funcion ingresar_paciente, para corregir el ingreso de datos de fecha, segun lo lee MySQL
    Fecha = datetime.strptime(Fecha, '%d-%m-%Y').strftime('%Y-%m-%d')

    # Verificar si el paciente existe
    cursor.execute("SELECT * FROM pacientes WHERE DNI=%s", (id_paciente,))
    paciente = cursor.fetchone()
    if paciente is None:
        print()
        print("Paciente no encontrado.")
        print()
        return

    cursor.execute("INSERT INTO turnos (id_turno, id_paciente, id_medico, Fecha, Hora, Especialidad, Motivo_consulta) VALUES (NULL, %s, %s, %s, %s, %s, %s)",
    (id_paciente, id_medico, Fecha, Hora, Especialidad, Motivo_consulta))
    conn.commit()
    print()
    print("Turno creado con exito!")
    print()
    conn.close()
```

Realiza verificaciones sobre **paciente**, **médico**, **fecha y hora**, para evaluar si existen (paciente y médico) y si no se encuentra duplicado (fecha y hora) y finalmente se verifica si la **fecha de la consulta** es anterior a la fecha actual.

```
# Insertar el nuevo turno en la tabla turnos
cursor.execute("INSERT INTO turnos (id_turno, id_paciente, id_medico, Fecha, Hora, Especialidad, Motivo_consulta) VALUES (NULL, %s, %s, %s, %s, %s, %s)",
(id_paciente, id_medico, Fecha, Hora, Especialidad, Motivo_consulta))
conn.commit()
print()
print("Turno creado con exito!")
print()
conn.close()
```

Finalmente, si todas las verificaciones son **exitosas**, se crea un nuevo turno en la tabla de “turnos”. mediante ejecutar el comando SQL, usando “cursor.execute” y el respectivo commit.

Buscar y traer un paciente por DNI:

```
# funcion para buscar y traer un paciente en base al dato <DNI>, que funciona como su clave primaria
def buscar_paciente_por_dni(conn, cursor, DNI):
    try:
        cursor.execute("SELECT * FROM pacientes WHERE DNI=%s", (DNI,))
        paciente = cursor.fetchone()
        if paciente is None:
            print()
            print("No se encontró un paciente con el DNI proporcionado.")
            print()
        else:
            return paciente
    except mysql.connector.Error as err:
        print()
        print(f"Error: {err}")
        print()
```

En esta función de **consulta**, se busca a un paciente en base a su **DNI**, la cual funciona como su **Clave Primaria** o “**Primary Key**” y se muestra en la consola.

Consta además del bloque “try/except”, de un “if” en caso de que no exista el paciente en la base de datos

Buscar y traer todos los médicos por especialidad:

```
# Funcion para traer a todos los medicos correspondiente de una determinada especialidad
def buscar_medicos_por_especialidad(conn, cursor, Especialidad):
    try:
        cursor.execute("SELECT * FROM medicos WHERE Especialidad=%s", (Especialidad,))
        medicos = cursor.fetchall()
        if not medicos:
            print()
            print("No se encontraron médicos con la especialidad proporcionada.")
            print()
        else:
            return medicos
    except mysql.connector.Error as err:
        print()
        print(f"Error: {err}")
        print()
```

Misma estructura que la **consulta** anterior pero utilizando como **filtro** el atributo: **especialidad** dentro de la tabla “medicos”., haciendo uso de comando <SELECT * FROM medicos>, seleccionado de la tabla medicos, todos los valores del médico cuyo atributo <Especialidad> sea igual al atributo ingresado.

Buscar y traer a todos los médicos existentes:

```
# funcion para buscar y traer todos los medicos registrados
def buscar_medicos(conn, cursor):
    cursor.execute("SELECT * FROM medicos")
    medicos = cursor.fetchall()
    return medicos
```

De nuevo, hay consultas como “**Buscar y traer a todos los pacientes existentes**” o “**Buscar y traer todos los turnos registrados**”, pero al tener una estructura similar a la anterior, no se incluye imagen.

Borrar datos de las tablas:

```
# funcion para eliminar/borrar registro de un turno de la respectiva tabla
def cancelar_turno(conn, cursor, id_turno):
    cursor.execute("SELECT * FROM turnos WHERE id_turno=%s", (id_turno,))
    turno = cursor.fetchone()
    if turno is None:
        print()
        print("Turno no disponible/encontrado.")
        print()
        return

    cursor.execute("DELETE FROM turnos WHERE id_turno=%s", (id_turno,))
    conn.commit()
    print("Turno cancelado con exito!")
```


```
# funcion para eliminar/borrar paciente del registro
def eliminar_paciente(conn, cursor, DNI):
    cursor.execute("SELECT * FROM pacientes WHERE DNI=%s", (DNI,))
    paciente = cursor.fetchone()
    if paciente is None:
        print()
        print("Paciente no encontrado.")
        print()
        return

    cursor.execute("DELETE FROM pacientes WHERE DNI=%s", (DNI,))
    conn.commit()
    print()
    print("Paciente eliminado con éxito!")
    print()

# funcion para eliminar/borrar medico del registro
def eliminar_medico(conn, cursor, DNI):
    cursor.execute("SELECT * FROM medicos WHERE DNI=%s", (DNI,))
    medico = cursor.fetchone()
    if medico is None:
        print()
        print("Médico no encontrado.")
        print()
        return

    cursor.execute("DELETE FROM medicos WHERE DNI=%s", (DNI,))
    conn.commit()
    print()
    print("Médico eliminado con éxito!")
    print()
```

Similar con el generador de turno, primero realiza una verificación para comprobar si existe el objeto indicado, y a continuación se ejecuta el comando que toma y elimina todos los valores de la tupla en la que se encuentra el valor igual al ingresado, eliminando así el objeto indicado.

En el archivo  [menu.py](#) encontramos, lo que se podría decir, la parte visual del programa, mostrando los diferentes menús en consola. Al ser tantos submenús, solo se presentan los principales. Algunos de estos son:

Menú principal:

```
def menuPrincipal(self):
    while True:
        print("Bienvenido al menu principal")
        print("1 - Consulta sobre Pacientes")
        print("2 - Consulta sobre Medicos")
        print("3 - Consulta sobre Turnos")
        print("4 - Salir")
        print()

        opcion = input("Elija una opcion: ")

        if opcion == '1':
            self.menuPaciente()
        elif opcion == '2':
            self.menuMedico()
        elif opcion == '3':
            self.menuTurno()
        elif opcion == '4':
            print("Saliendo del programa...")
            break
        else:
            print("Opción no válida. Por favor, intente de nuevo.")
```

SubMenú Paciente:

```
# Sub Menu relacionado a las opciones de paciente.
def menuPaciente(self):
    while True:
        print("1 - Adicionar paciente")
        print("2 - Buscar un paciente")
        print("3 - Eliminar paciente")
        print("4 - Exhibir pacientes registrados")
        print("9 - Salir")

        opcion = input("Elija una opcion: ")
```

Opción Adicionar paciente:

```
# Se solicita ingresar todos los valores necesarios para crear un paciente. Tener en cuenta que uno de los valores es una fecha.
if opcion == "1":
    Nombre = input("Digite Nombre del paciente: ")
    Apellido = input("Digite Apellido del paciente: ")
    DNI = input("Digite DNI del paciente: ")
    Fecha_Nacimiento = input("Digite Fecha de Nacimiento del paciente (DD-MM-AAAA): ")
    Telefono1 = input("Digite Telefono1 del paciente: ")
    Telefono2 = input("Digite Telefono2 del paciente, en caso de no poseer coloque un guion: ")
    Email = input("Digite Email del paciente: ")
    Provincia = input("Ingrese provincia: ")
    Ciudad = input("Ingrese ciudad/localidad: ")
    CP = input("Ingrese el codigo postal: ")
    Calle = input("Ingrese la calle: ")
    Numero = input("Digite altura de la direccion: ")
    Depto = input("Digite piso/departamento/lote: ")
    agregar_paciente(self.db, self.cursor, Nombre, Apellido, DNI, Fecha_Nacimiento, Telefono1, Telefono2, Email, Provincia, Ciudad, Calle, CP, Numero, Depto)
    print("-----")
```

Consultas utilizando funciones del archivo funciones.py:





```
# Esta opcion llama a la fucion para buscar un paciente, mediante su DNI
elif opcion == "2":
    DNI = input("Digite DNI del paciente a buscar: ")
    paciente = buscar_paciente_por_dni(self.db, self.cursor, DNI)
    if paciente:
        print(f"Paciente encontrado: DNI: {paciente[0]}, Nombre: {paciente[1]}, Apellido: {paciente[2]}, Fecha de Nacimiento: {paciente[3]}")
    else:
        print("No se encontró un paciente con ese DNI.")
    print("-----")

# Y esta funcion permite eliminarlos
elif opcion == "3":
    DNI = input("Digite DNI del paciente a eliminar: ")
    eliminar_paciente(self.db, self.cursor, DNI)
    print("-----")

    # Trae la funcion para buscar pacientes en la base.
elif opcion == "4":
    pacientes = buscar_pacientes(self.db, self.cursor)
    print("Pacientes registrados:")
    for paciente in pacientes:
        print(f"DNI: {paciente[0]}, Nombre: {paciente[1]}, Apellido: {paciente[2]}, Fecha de Nacimiento: {paciente[3]}")
    print("-----")
```

Esta estructura se aplica de forma similar en los submenús “Médicos” y “Turnos”, pero utilizando sus propias funciones y consultas.


Se ve en cada caso se otorgan los valores self.db y y self.cursor tomados y definidos dentro del archivo main.py.

El archivo  **main.py** es el documento principal del programa que da inicio al mismo y ejecuta en orden el resto de los elementos que lo conforman como el  **database.py**, que establece la conexión con la base de datos y el  **menu.py** que a su vez llama a las funciones definidas en el apartado  **funciones.py** según la opción que se marque.

```
def main():
    # Crear una instancia de la base de datos
    db, cursor = conectar_db()
    if db:
        print("Conexión establecida exitosamente")
        menu = Menu(db, cursor)
        menu.menuPrincipal()
        db.close()
    else:
        print("No se pudo establecer la conexión a la base de datos")

if __name__ == "__main__":
    main()
```

La función main llama primero a la función “conectar_bd” definida en database.py. Esta devuelve una conexión a la base de datos, y un cursor, y los almacena en las variables “db” y “cursor” respectivamente. Luego mediante un “if” verifica si se establece la conexión con la base de datos correctamente, sino, a un aviso. Si se concreta la conexión lo indica mediante un mensaje en la consola, luego crea una instancia de la clase “Menu” pasando las variables “cursor” y “db” como argumento. Siguiendo llama al método “menuPrincipal” de la instancia “menu” y con el “db.close” cierra la conexión una vez que se concreta la interacción con el menu

El archivo  **PROYECTO ISPC.sql** contiene comandos SQL para crear la estructura inicial de la base de datos junto con ejemplos de prueba para cada tabla.

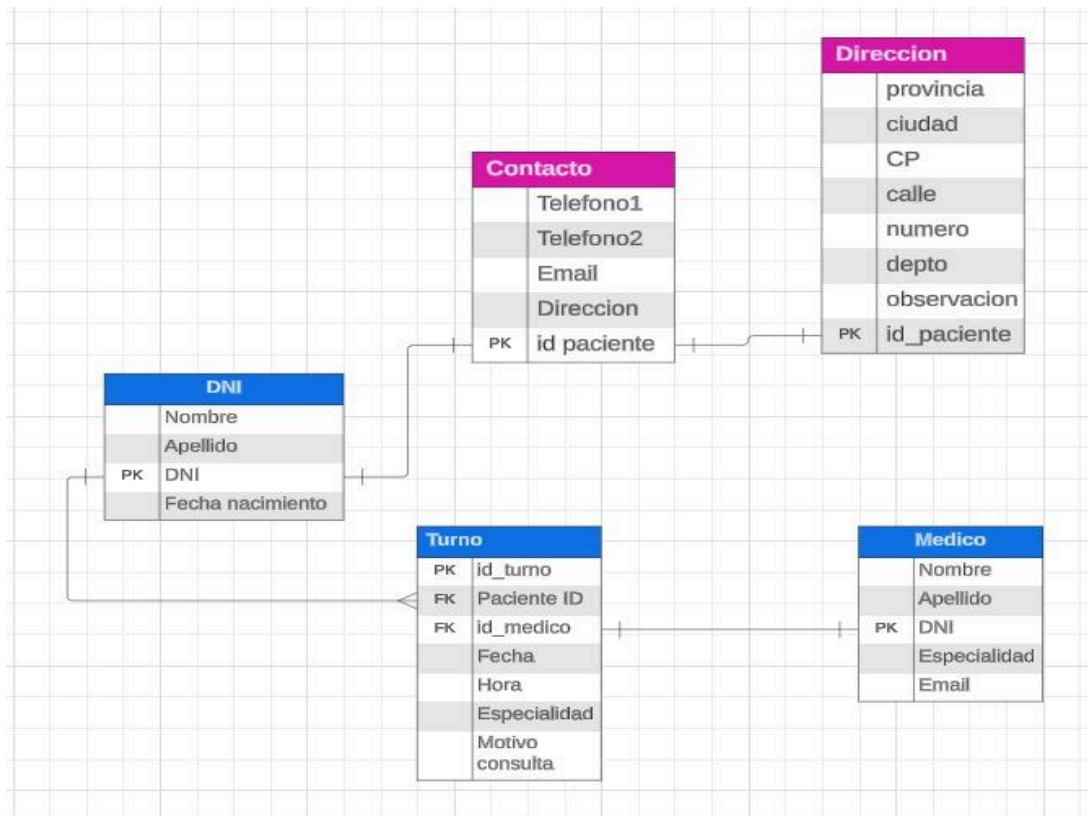
```
Run | New Tab | Copy
CREATE TABLE IF NOT EXISTS pacientes (
    DNI VARCHAR(20) PRIMARY KEY,
    Nombre VARCHAR(50) NOT NULL,
    Apellido VARCHAR(50) NOT NULL,
    Fecha_Nacimiento DATE NOT NULL,
    Telefono1 VARCHAR(20) NOT NULL,
    Telefono2 VARCHAR(20),
    Email VARCHAR(100),
    Provincia VARCHAR(50),
    Ciudad VARCHAR(50),
    Calle VARCHAR(100),
    CP VARCHAR(10),
    Numero VARCHAR(10),
    Depto VARCHAR(10)
);
```

```
Run | New Tab
INSERT INTO pacientes (DNI, Nombre, Apellido, Fecha_Nacimiento, Telefono1, Telefono2, Email, Provincia, Ciudad, Calle, CP, Numero, Depto) VALUES
('12345678', 'Juan', 'Pérez', '1980-01-01', '1234567890', '0987654321', 'juan.perez@example.com', 'Córdoba', 'Córdoba', 'Calle Falsa', '5000', '123', 'A'),
('87654321', 'María', 'González', '1990-01-01', '0987654321', '1234567890', 'maria.gonzalez@example.com', 'Córdoba', 'Córdoba', 'Calle Verdadera', '5000', '321', 'B');

Run | New Tab
INSERT INTO medicos (DNI, Nombre, Apellido, Especialidad, Email) VALUES
('11223344', 'Carlos', 'Rodríguez', 'Cardiología', 'carlos.rodriguez@example.com'),
('44332211', 'Ana', 'Martínez', 'Neurología', 'ana.martinez@example.com');

Run | New Tab
INSERT INTO turnos (id_paciente, id_medico, Fecha, Hora, Especialidad, Motivo_consulta) VALUES
('12345678', '11223344', '2024-06-10', '10:00:00', 'Cardiología', 'Control rutinario'),
('87654321', '44332211', '2024-06-11', '11:00:00', 'Neurología', 'Dolor de cabeza persistente');
```

Modelo relacional normalizado - Diagrama ER



RELACIONES:

Tabla **DNI**:

→ Relación con **Turno**: “**DNI (DNI)**” → “**Turno (Paciente ID)**”

Tipo de relación: Uno a muchos (Un paciente puede tener múltiples turnos)

→ Relación con **Contacto**: “**DNI (DNI)**” → “**Contacto (id_paciente)**”

Tipo de relación: Uno a uno (Un paciente tiene un solo contacto asociado)

→ Relación con **Direccion**: “**DNI (DNI)**” → “**Direccion (id_paciente)**”

Tipo de relación: Uno a uno (Un paciente tiene una sola dirección asociada)

Tabla **Contacto**:

→ Relación con **DNI**: “**Contacto (id_paciente)**” → “**DNI (DNI)**”

Tipo de relación: Uno a uno (Cada contacto está asociado con un solo paciente)

Tabla **Direccion**:

→ Relación con **DNI**: “**Direccion (id_paciente)**” → “**DNI (DNI)**”

Tipo de relación: Uno a uno (Cada dirección está asociada con un solo paciente)

Tabla **Turno**:

→ Relación con **DNI**: “**Turno (Paciente ID)**” → “**DNI (DNI)**”

Tipo de relación: Muchos a uno (Muchos turnos pueden estar asociados con un solo paciente)

→ Relación con **Medico**: “**Turno (id_medico)**” → “**Medico (DNI)**”

Tipo de relación: Muchos a uno (Muchos turnos pueden estar asociados con un solo médico)

Tabla **Medico**:

→ Relación con **Turno**: “**Medico (DNI)**” → “**Turno (id_medico)**”

Tipo de relación: Uno a muchos (Un médico puede tener múltiples turnos asociados)

NORMALIZACIÓN:

1FN: Cada atributo en las tablas es indivisible

2FN: Ningún atributo en las tablas depende parcialmente de algún otro atributo

3FN: Y todo atributo de cada tabla está sujeto y depende únicamente a la clave primaria.

Aspectos Éticos y Profesionales:

¿Qué dicta la Ley 7642 de la provincia de Córdoba?

La **Ley 7642** de la provincia de Córdoba fue sancionada el **25 de noviembre de 1987**. Esta establece las condiciones para el ejercicio profesional en el ámbito de las ciencias informáticas en la provincia. La ley creó el **Consejo Profesional de Ciencias Informáticas de la Provincia de Córdoba (CPCIPC)**, que es el organismo encargado de **regular, matricular y supervisar** la actividad de los profesionales de este campo. Además, define un código de ética y los deberes tanto para los profesionales como para los clientes.

La ética profesional en el desarrollo de software es fundamental para garantizar que los productos creados sean **seguros, fiables y respetuosos** con los usuarios y la sociedad en general. Los desarrolladores tienen la responsabilidad de crear aplicaciones que no solo funcionen correctamente, sino que también protejan la **privacidad** y los **datos** de los usuarios. La falta de ética puede llevar a la creación de software con vulnerabilidades que pueden ser **explotadas**, resultando en pérdidas **económicas**, daños en la **reputación** de empresas y riesgos para la **seguridad personal** o hasta **nacional**.

La ética profesional es crucial para evitar prácticas desleales como el **plagio** o el uso de **código sin licencia** adecuada. En un campo tan dinámico y en constante evolución como el desarrollo de software, mantener altos estándares éticos ayuda a orientar las **decisiones** y **acciones** de los profesionales, fomentando un entorno de trabajo justo y una industria tecnológica **sostenible** y **equitativa**.

Nosotros creemos que el **CPCIPC** es un estatuto fundamental para garantizar que los profesionales del sector mantengan estándares éticos y técnicos elevados,

asegurando así la calidad y la confiabilidad de los servicios informáticos creados en la provincia.

A su vez, también proporciona un marco regulador que fomenta la actualización continua y la responsabilidad profesional estableciendo una base para la certificación y acreditación de competencias, lo cual es crucial para distinguir a los profesionales calificados y garantizar que los proyectos y desarrollos tecnológicos sean llevados a cabo por individuos con la formación adecuada y experiencia relevante.

Principios de la Ley 7642 y aplicación en nuestro proyecto:

Consentimiento Informado:

Aplicación: Antes de recopilar y almacenar cualquier dato personal o médico, se debe obtener el consentimiento explícito de cada paciente. Este consentimiento debe ser informado, es decir, los pacientes deben saber qué datos se recopilan, con qué propósito y cómo se utilizarán.

Implementación: Integrar en el programa una funcionalidad que permita presentar un formulario de consentimiento digital, que los pacientes deben leer y aceptar antes de proceder con el registro de sus datos.

Finalidad Específica:

Aplicación: Los datos personales deben ser recolectados con fines específicos, explícitos y legítimos, y no deben ser utilizados para propósitos incompatibles con estos fines.

Implementación: Definir y documentar claramente los propósitos del programa y asegurarse de que todas las funcionalidades del sistema se alineen con estos fines.

Calidad de los Datos:

Aplicación: Los datos deben ser exactos, adecuados, pertinentes y no excesivos en relación con el fin para el que se recogen.

Implementación: Establecer mecanismos para verificar la exactitud de los datos ingresados y permitir actualizaciones regulares. Limitar la recopilación de datos a solo aquellos necesarios para la gestión y monitoreo de los pacientes.

Seguridad de los Datos:

Aplicación: Se deben implementar medidas técnicas y organizativas adecuadas para garantizar la seguridad de los datos personales y evitar su alteración, pérdida, tratamiento o acceso no autorizado.

Implementación: Incluir en el programa medidas de seguridad como encriptación de datos, autenticación de usuarios, registros de acceso y políticas de control de acceso.

Transparencia:

Aplicación: Los pacientes tienen derecho a conocer quién está tratando sus datos, con qué propósito y a quiénes podrían ser comunicados.

Implementación: Proveer una sección en el programa donde se detallen las políticas de privacidad y manejo de datos, y mantener a los pacientes informados sobre cualquier cambio en estas políticas.

Acceso y Rectificación:

Aplicación: Los pacientes tienen derecho a acceder a sus datos y a solicitar la rectificación de los mismos si son incorrectos o incompletos.

Implementación: Incluir funcionalidades en el programa que permitan a los pacientes, o al Dr. Javier en su representación, acceder a sus datos y corregir cualquier información incorrecta.

Responsabilidad Proactiva:

Aplicación: El responsable del tratamiento de datos debe ser capaz de demostrar que está cumpliendo con las obligaciones de la ley.

Implementación: Documentar todas las políticas y procedimientos relacionados con la protección de datos y mantener registros detallados de cómo se gestionan los datos. Realizar auditorías periódicas para asegurar el cumplimiento continuo.