

# Trabajo Final



Tópicos de Ingeniería de Software II  
UNLP  
Magister en Ingeniería de Software

Profesores:  
Pons, Claudia  
Pace, Andres Diaz  
Urbieta, Matias

Integrantes:  
Murdolo, Juan  
Jurado, Abraham

<b>Enunciado</b>	<b>1</b>
Entregables	2
<b>Introducción</b>	<b>3</b>
<b>Definición del modelo</b>	<b>3</b>
<b>Flask</b>	<b>9</b>
Directorio App	9
__init__.py	9
schemas.py	10
db.py	10
Directorio Models	11
__init__.py	11
predict.py	11
user.py	12
Directorio resources	12
freemium y premium.py	13
user.py	14
Directorio Instance	15
Directorio Migrations	15
<b>Pruebas</b>	<b>16</b>
<b>Conclusiones</b>	<b>21</b>

## Enunciado

Este trabajo integrador consiste en el desarrollo de un servicio Web (API) que expone un modelo de red neuronal para clasificar los parámetros del servicio:

El modelo diagnóstica riesgo cardíaco (si el paciente tiene o no-tiene riesgo cardíaco) a partir de los datos clínicos del paciente (nivel de colesterol, presión arterial, nivel de azúcar en sangre, edad, sobrepeso y tabaquismo) .

El servicio solo deberá poder ser invocado por clientes de nuestra plataforma y por lo tanto deberán pasar una API Key. Las invocaciones (request) HTTP deberá contar con el header HTTP 'Authorization' indicando la API key. Si la API key no se encuentra en la invocación, ésta deberá ser rechazada.

GET /service

Authorization: <API Key generada>

```
{
  "nivel_colesterol": 2.00,
  "presion_arterial":1.20,
  "azucar":1.00
  "edad":40,
  "sobrepeso":1,
```

```
    "tabaquismo":0  
}
```

Aclaración. los rangos de los datos son:

```
colesterol = (1.0, 3.0)  
presion = (0.6, 1.8)  
azucar= (0.5, 2.0)  
edad = (0,99)  
sobrepeso 0 o 1, donde 1 representa presencia de sobrepeso.  
tabaquismo 0 o 1, donde 1 representa presencia de tabaquismo.
```

Existen dos tipos de cuentas que restringen la cantidad de solicitudes HTTP por minuto que el sistema está autorizado a resolver por minuto:

- FREEMIUM; 5 solicitudes por minuto (RPM).
- PREMIUM: 50 solicitudes por minuto (RPM).

El servicio deberá satisfacer los siguientes requerimientos:

- Deberá correr la red neuronal entrenada previamente.
- Todas las invocaciones que reciba el servicio deberán ser controladas verificando dos aspectos:
  - Autorización. A partir de la API key, se verifica si exista registrada la API key en la base de datos del sistema.
  - Limitación. De acuerdo a la suscripción del cliente tiene una limitación de invocaciones por segundo: FREEMIUM y PREMIUM.
- Cada solicitud recibida deberá ser registrada en la bitácora (log). Capturando el tiempo que tomo para procesar el requerimiento HTTP de diagnostico: iniciar el timer cuando se recibe la solicitud HTTP, procesar la autenticación de la key, correr la red neuronal, registrar el resultado en la bitácora, y retornar la respuesta.
- Para datos de solo lectura y de poca volatilidad, se espera que se implemente cache.

## Entregables

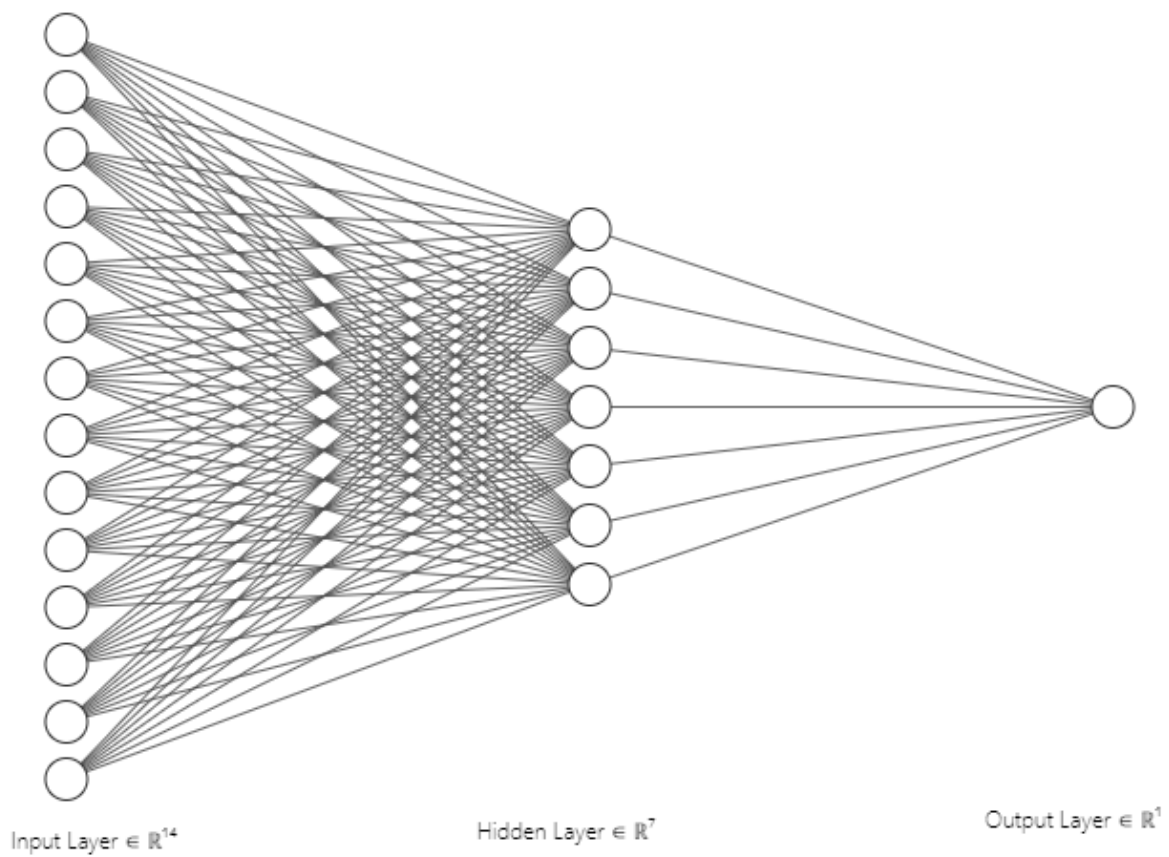
La solución deberá contar con los siguientes entregables:

- Instructivo para correr el proceso de entrenamiento del modelo de ML
- Informe de diseño donde se presenten diagramas, aclaraciones sobre los requerimientos y toma de decisiones.
- Bateria de test HTTP para probar el funcionamiento de los servicios.

# Introducción

## Definición del modelo

A continuación presentamos el diagrama con la arquitectura de la red neuronal que entrenamos para resolver la clasificación:



Si imprimimos un resumen del modelo secuencial obtenemos:

```

Model: "sequential"
-----
Layer (type)                Output Shape              Param #
-----
dense (Dense)                (None, 14)                112
dense_1 (Dense)              (None, 7)                 105
dense_2 (Dense)              (None, 1)                  8
-----
Total params: 225 (900.00 Byte)
Trainable params: 225 (900.00 Byte)
Non-trainable params: 0 (0.00 Byte)

```

Este modelo se encuentra en el archivo **0\_ai/entrenamiento.py**, el código tiene como objetivo crear y entrenar una red neuronal que prediga el riesgo cardíaco de los pacientes a partir de sus datos clínicos, a continuación comentamos las líneas más importantes del archivo:

```

X = data.drop('riesgo_cardiaco', axis=1)
Y = data['riesgo_cardiaco']

```

Aquí separamos las características (X) de la variable objetivo (Y). X contiene todas las columnas excepto 'riesgo\_cardiaco', mientras que Y contiene sólo la columna 'riesgo\_cardiaco'. En este caso, el conjunto de datos se encuentra en un archivo CSV llamado datos\_de\_pacientes\_5000.csv que debe moverse de la carpeta 0\_ai a la carpeta 0\_ai/RedesNeuronales. El conjunto de datos contiene 5000 registros, cada uno con 7 variables predictoras y una variable de salida.

```

scaler = MinMaxScaler()
X = scaler.fit_transform(X)

```

Como los datos recibidos se encuentran en diferentes escalas, es necesario normalizar para asegurarnos que todas las características estén en la misma escala, lo que ayuda al modelo a converger más rápido durante el entrenamiento. Se normalizan los datos en X utilizando MinMaxScaler para que todas las características estén en el rango [0, 1].

```

X_train, X_test, Y_train, Y_test = train_test_split(
    X, Y, test_size=0.2, random_state=42)

```

Se dividen los datos normalizados en conjuntos de entrenamiento (X\_train, Y\_train) y prueba (X\_test, Y\_test) utilizando train\_test\_split. Esto permite evaluar la capacidad del modelo en datos que no ha visto durante el entrenamiento, un 20% de los datos para prueba y fijando una semilla aleatoria (42) para reproducibilidad.

```
model = Sequential()

model.add(Dense(14, input_dim=7, activation="relu"))
model.add(Dense(7, activation="relu"))
model.add(Dense(1, activation="sigmoid"))
```

Después de muchas sesiones de entrenamiento probando diferentes tipos de configuraciones, consideramos que el siguiente modelo se acerca bastante a la solución que estábamos buscando.

Creamos una red neuronal de tipo **Sequential**, que representa un modelo de red neuronal como una secuencia de capas.

La red neuronal tiene tres capas:

- Una capa oculta con 14 neuronas: Esta es la primera capa de la red. Con el **input\_dim=7** se especifica que esta capa espera datos de entrada de 7 dimensiones (el número de columnas del DataFrame X) y usará la función de activación ReLU, que es una función no lineal que devuelve el máximo entre 0 y la entrada.
- Otra capa oculta con 7 neuronas: otra capa densa al modelo, con 7 neuronas y la misma función de activación ReLU. Esta capa recibirá como entrada la salida de la capa anterior.
- Una capa de salida con una neurona: la activación utilizada en las capas ocultas es relu, mientras que la activación utilizada en la capa de salida es **sigmoid**. La función de activación sigmoide es una función no lineal que devuelve un valor entre 0 y 1, representando la probabilidad de que el paciente tenga riesgo cardíaco.

```
model.compile(loss='binary_crossentropy',
              optimizer='adam', metrics=['accuracy'])
```

En este caso, se utiliza el optimizador **Adam**, que es un optimizador muy utilizado en el aprendizaje automático, ya que es un algoritmo de descenso de gradiente estocástico adaptativo que ajusta la tasa de aprendizaje de cada parámetro según su gradiente.

La función de pérdida **binary\_crossentropy** se utiliza para calcular la pérdida del modelo, es una medida de la diferencia entre la probabilidad predicha por la red y la probabilidad real de la clase.

Las métricas **accuracy** se utilizan para evaluar el rendimiento del modelo, lo que nos va a permitir saber cuál es la proporción de predicciones correctas sobre el total de predicciones

```
model.summary()
```

Usamos el método **summary** para imprimir un resumen de la arquitectura de la red neuronal, mostrando el número y el tipo de capas, el número de parámetros y la forma de las entradas y salidas.

```
history = model.fit(X_train, Y_train, epochs=150, batch_size=8,
```

```
validation_data=(X_test, Y_test))
```

En este caso, el modelo se entrena durante 150 épocas, es decir, es el número de veces que la red pasa por todo el conjunto de datos. El tamaño de lote de 8 datos, que es el número de muestras que la red procesa antes de actualizar sus pesos.

Especificamos los datos de validación, que son los datos de prueba (X\_test y Y\_test), que se usarán para evaluar el rendimiento de la red en cada época. Guardamos el resultado del entrenamiento en una variable llamada **history**, que es un objeto de tipo History que contiene información sobre la pérdida y la precisión de la red en cada época

```
test_loss = model.evaluate(X_test, Y_test)
print(test_loss)
```

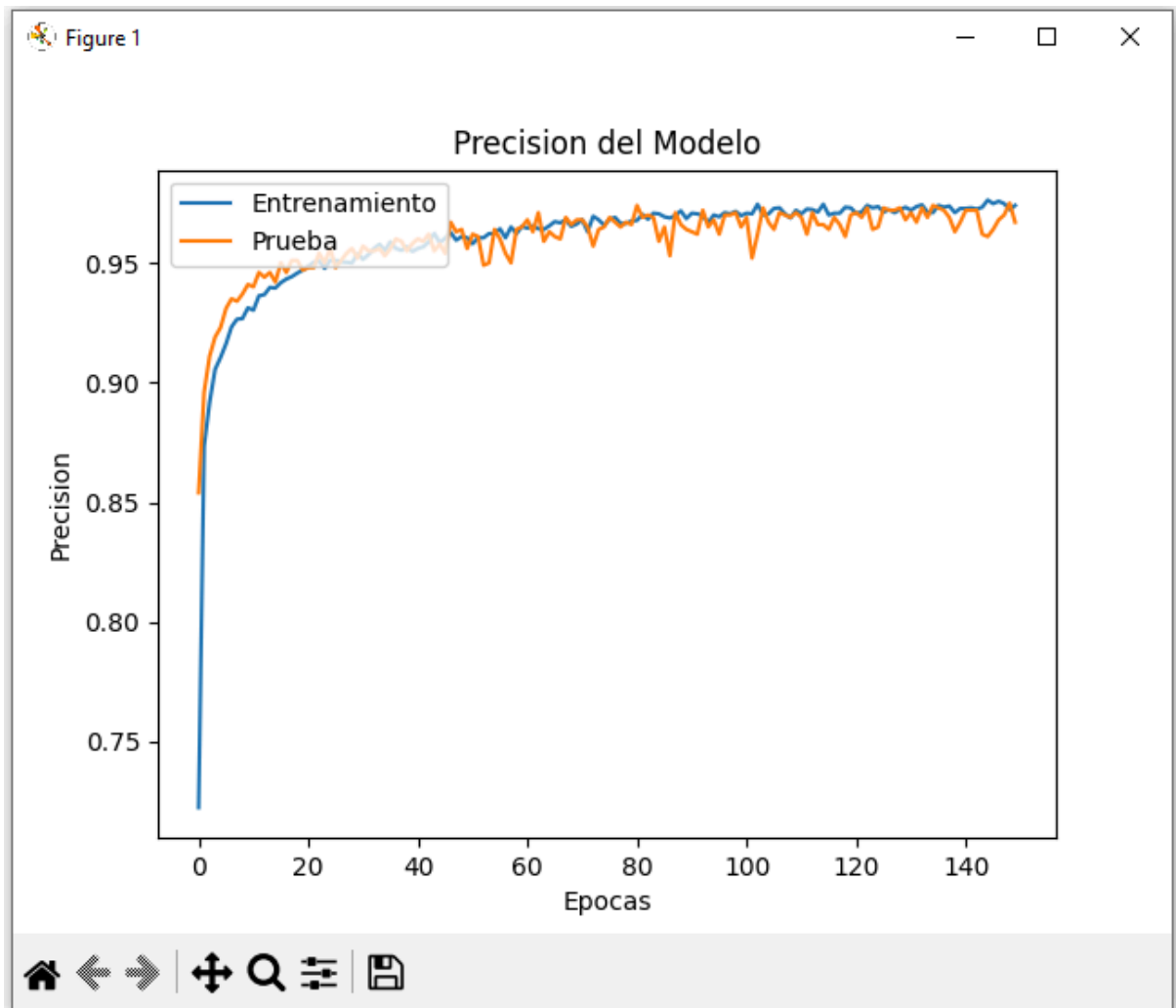
Usamos el método evaluate del objeto model para evaluar la red neuronal con los datos de prueba (X\_test y Y\_test), devolviendo la pérdida y la precisión de la red. Guardamos el resultado en una variable llamada test\_loss, que es una lista de dos elementos.

También agregamos líneas de código para visualizar la precisión y la pérdida del modelo durante el entrenamiento y la evaluación. La precisión se define como la proporción de datos correctamente clasificados. La pérdida se define como una medida de la diferencia entre las predicciones del modelo y los valores reales.

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Precisión del Modelo')
plt.ylabel('Precision')
plt.xlabel('Epocas')
plt.legend(['Entrenamiento', 'Prueba'], loc='upper left')
plt.show()
```

Primero, se recuperan las series temporales de la precisión del modelo en el conjunto de entrenamiento y en el conjunto de prueba. Estas series temporales se almacenan en los atributos history.history['accuracy'] y history.history['val\_accuracy'].

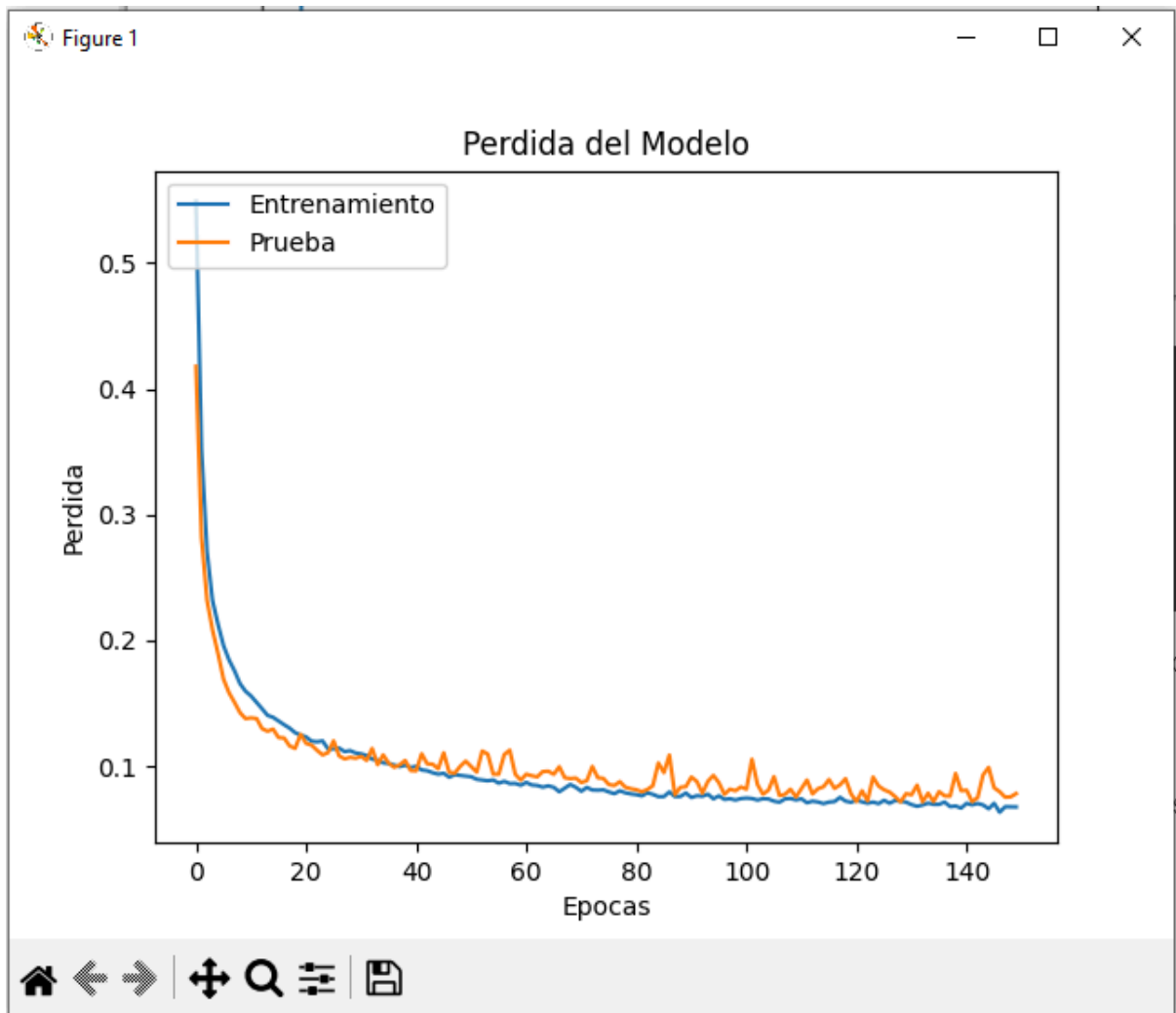
La primera línea en naranja muestra la precisión del modelo en el conjunto de entrenamiento en función de las épocas de entrenamiento. La segunda línea en azul muestra la precisión del modelo en el conjunto de prueba en función de las épocas de entrenamiento.



```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Pérdida del Modelo')
plt.ylabel('Pérdida')
plt.xlabel('Epocas')
plt.legend(['Entrenamiento', 'Prueba'], loc='upper left')
plt.show()
```

Este código funciona de manera similar que el anterior, pero es para la pérdida del modelo.





Estas gráficas pueden ser útiles para evaluar el rendimiento del modelo durante las etapas de entrenamiento y evaluación. La precisión muestra qué tan bien se desempeña el modelo en términos de clasificación, mientras que la pérdida indica qué tan bien está aprendiendo el modelo: valores más bajos de pérdida son preferibles. Por ejemplo, si la precisión del modelo en el conjunto de prueba no mejora después de un cierto número de épocas, es posible que el modelo esté sobreajustado o subajustado.

```
print("Datos a predecir:")
print(X_train[:5])
print("-----")
result = model.predict(X_train[:5])
print("Resultados obtenidos:")
print(result)
print("Valores correctos:")
print(Y_train[:5])
```

Muestra los primeros 5 datos de entrenamiento, realiza predicciones con el modelo y compara con los valores reales de entrenamiento para esos 5 datos.

En la siguiente imagen mostramos el modelo funcionando:

```
Resultados obtenidos:
[[0.04388356]
 [0.04531754]
 [0.99888456]
 [0.9803798 ]
 [0.01596154]]
Valores correctos:
4227    0
4676    0
800     1
3671    1
4193    0
Name: riesgo_cardiaco, dtype: int64
```

Vemos como las predicciones sobre los primeros 5 elementos del conjunto de entrenamiento fue bastante buena. Esta comparación entre los resultados predichos y los valores reales ayuda a evaluar la precisión del modelo.

## Flask

Para el backend utilizamos Flask con una base de datos de tipo sqlite, ya que se ajusta de manera perfecta a lo que tenemos que hacer y permite que el archivo de base de datos sea pequeño, no utilizamos mongo por una cuestión de confort ya que nos sentíamos más cómodos utilizando sqlite.

Utilizamos la estructura típica de directorios de proyectos de Flask dentro de 1\_flask.

## Directorio App

Contiene los archivos esenciales: db.py, \_\_init\_\_.py, schemas.py, y los otros directorios .env, models, resources

### \_\_init\_\_.py

- Importaciones: Importa varios módulos y paquetes necesarios para construir una API con Flask, incluyendo Flask mismo, extensiones como Flask-JWT-Extended para autenticación JWT, Flask-Migrate para migraciones de base de datos, Flask-Smorest para definir endpoints de API, entre otros.
- Creación de la aplicación: La función create\_app() es el punto de entrada principal. Configura una instancia de Flask y realiza varias configuraciones como la base de datos (utilizando SQLAlchemy), el sistema de autenticación JWT (para identificar si el usuario es freemium o premium), y la carga del modelo neuronal TensorFlow.

- Configuración: Se establecen múltiples configuraciones para la API, incluyendo título, versión, URL base para la documentación de Swagger, la URI de la base de datos, y claves secretas para JWT.
- Inicialización de extensiones: Se inicializan y configuran las extensiones como la base de datos (usando SQLAlchemy y Flask-Migrate), JWTManager para la gestión de tokens JWT, y la configuración de límites de acceso usando Flask-Limiter.
- Creación de blueprints: Se registran blueprints (conjuntos de rutas y lógica relacionada) para diferentes recursos de la API, como usuarios, la entrada y lectura de datos de riesgo\_cardiaco, y las limitaciones relacionadas con el tipo de usuario, separándolos en freemium y premium.

## schemas.py

En este código se emplea la biblioteca marshmallow, que proporciona un modo de serializar y deserializar objetos Python. Aquí, se están definido esquemas (schemas) que describen la estructura y validaciones para diferentes tipos de datos que se usarán en la aplicación.

- PredictSchema: Este esquema está diseñado para validar datos relacionados con la predicción de riesgo cardíaco. Define campos como nombre, presión arterial, colesterol, azúcar en sangre, edad, sobrepeso y tabaquismo. Cada campo tiene diferentes validaciones, como valores predeterminados, requerimientos de longitud mínima, y requerimientos de tipo de dato.
- PredictFinishedSchema: Este es un esquema que hereda de PredictSchema y agrega un campo adicional llamado riesgoCardiaco, que se espera en la respuesta final de la predicción, este es el Schema que se va a guardar en la base de datos al hacer la predicción.
- UserSchemaBasic: Este esquema es para datos básicos del usuario, incluyendo nombre de usuario y contraseña. La contraseña solo se carga (load\_only) pero no se muestra en la salida serializada (dump\_only).
- UserRegisterSchema: Hereda de UserSchemaBasic y agrega un campo de correo electrónico. Este esquema se utilizaría para registrar nuevos usuarios, validando tanto el nombre de usuario como el correo electrónico.
- UserSchema: Es un esquema más completo que se usa para mostrar los detalles completos del usuario, incluyendo un ID, correo electrónico, tipo de usuario, etc. Este esquema podría ser útil para mostrar detalles de usuario después del registro o para otros fines en la aplicación.

Estos esquemas proporcionan una estructura definida y validaciones para diferentes tipos de datos, desde detalles de predicción de riesgo cardíaco hasta detalles de usuario, permitiendo la serialización y validación de estos datos en la aplicación.

## db.py

Este código importa y crea una instancia de SQLAlchemy, una extensión de Flask utilizada para interactuar con bases de datos relacionales a través de modelos de objetos Python. SQLAlchemy simplifica las operaciones de base de datos al permitir a los desarrolladores trabajar con objetos Python en lugar de escribir consultas SQL directamente.

La instancia creada (db) se utiliza para definir modelos de datos (tablas de base de datos) y realizar operaciones como consultas, inserciones, actualizaciones y eliminaciones en la base de datos mediante métodos y atributos proporcionados por SQLAlchemy. Esta instancia se usa para realizar la vinculación entre la aplicación Flask y la base de datos, permitiendo el acceso y la manipulación de los datos de una manera más orientada a objetos.

## Directorio Models

El directorio models se compone de 3 archivos un `__init__.py`, y los archivos que van a funcionar para que se generen las tablas en la base de datos `user.py`, `predict.py`.

El archivo `__init__.py` dentro de la carpeta models se ejecuta y, al contener las importaciones de los modelos (PredictModel y UserModel) desde los archivos respectivos (`predict.py` y `user.py`), hace que estos modelos estén disponibles directamente desde el paquete models. Esto facilita la importación y el uso de esos modelos desde otros lugares del proyecto sin necesidad de importarlos desde archivos específicos (`predict.py` y `user.py`).

### `__init__.py`

En este contexto específico, el archivo `__init__.py` dentro de la carpeta models se utiliza para agrupar y exponer los modelos de datos (como PredictModel y UserModel) que se encuentran en archivos individuales (`predict.py` y `user.py`) dentro de la misma carpeta.

### `predict.py`

- `from ..db import db`: Importa la instancia de SQLAlchemy (db) desde un módulo superior (`..db`). Esto asume que hay un archivo `db.py` en un directorio superior al que se está trabajando, y desde allí se está importando la instancia db.
- `class PredictModel(db.Model)`: Define una clase llamada PredictModel que hereda de `db.Model`. Por convención en Flask-SQLAlchemy, los modelos de datos generalmente heredan de `db.Model`, lo que indica que estos modelos representan tablas de la base de datos.
- `__tablename__ = "predict"`: Esta variable de clase establece el nombre de la tabla en la base de datos que este modelo representará. En este caso, el nombre de la tabla es "predict".
- Campos del modelo:
  - `id = db.Column(db.Integer, primary_key=True)`: Define un campo id como un entero que servirá como clave primaria en la tabla.
  - `nombre = db.Column(db.String(80))`: Define un campo nombre como una cadena de texto de hasta 80 caracteres.
  - `presionArterial = db.Column(db.Float)`: Campo para la presión arterial, almacenada como un número de punto flotante.
  - `colesterol = db.Column(db.Float)`: Campo para el nivel de colesterol, también almacenado como un número de punto flotante.
  - `azucar = db.Column(db.Float)`: Campo para el nivel de azúcar, almacenado como un número de punto flotante.

- `edad = db.Column(db.Integer)`: Campo para la edad, almacenada como un entero.
- `sobrepeso = db.Column(db.Boolean)`: Campo booleano para indicar si hay sobrepeso.
- `tabaquismo = db.Column(db.Boolean)`: Campo booleano para indicar si el individuo es fumador.
- `riesgoCardiaco = db.Column(db.Boolean)`: Campo booleano para indicar el riesgo cardiaco.

Este modelo define la estructura de una tabla llamada "predict" en la base de datos, con columnas específicas correspondientes a los campos especificados en el modelo.

#### user.py

- `from ..db import db`: Importa la instancia de SQLAlchemy (db) desde un módulo superior (..db). Similar al ejemplo anterior, esto implica que se está importando la instancia db desde un directorio superior.
- `class UserModel(db.Model)::` Define una clase llamada UserModel que hereda de db.Model. Al igual que en el ejemplo anterior, esto indica que este modelo representa una tabla en la base de datos.
- `__tablename__ = "users"`: Establece el nombre de la tabla en la base de datos que este modelo representará. En este caso, el nombre de la tabla es "users".
- Campos del modelo:
  - `id = db.Column(db.Integer, primary_key=True)`: Define un campo id como un entero que actuará como clave primaria en la tabla.
  - `username = db.Column(db.String(80), unique=True, nullable=False)`: Campo para el nombre de usuario, almacenado como una cadena de texto de hasta 80 caracteres. `unique=True` indica que cada nombre de usuario debe ser único en la base de datos. `nullable=False` especifica que este campo no puede ser nulo.
  - `password = db.Column(db.String(80), nullable=False)`: Campo para la contraseña del usuario, almacenada como una cadena de texto de hasta 80 caracteres. `nullable=False` indica que este campo no puede ser nulo.
  - `email = db.Column(db.String(80), unique=True, nullable=False)`: Campo para el correo electrónico del usuario, almacenado como una cadena de texto de hasta 80 caracteres. `unique=True` asegura que cada dirección de correo electrónico sea única en la base de datos. `nullable=False` especifica que este campo no puede ser nulo.
  - `type = db.Column(db.String(80), nullable=False)`: Campo para el tipo de usuario, almacenado como una cadena de texto de hasta 80 caracteres. `nullable=False` indica que este campo no puede ser nulo.

#### Directorio resources

Tiene 3 archivos `freemium.py`, `premium.py` y `user.py`

## freemium y premium.py

Estos dos bloques de código son bastante similares en su estructura y funcionalidad. Ambos definen endpoints para la API utilizando la extensión Blueprint de flask\_smorest. La principal diferencia radica en las restricciones de acceso basadas en el tipo de usuario (freemium o premium) y la limitación de acceso a las rutas.

Aquí está el desglose de cada uno:

### Blueprint "Freemium":

- Limitación de acceso: Las rutas están limitadas para usuarios del tipo "freemium" con una restricción en el número de solicitudes.
- Rutas:
  - /freemium/predict: Permite realizar operaciones de predicción para usuarios "freemium".
  - /freemium/predictById/<predict\_id>: Permite obtener predicciones específicas por ID para usuarios "freemium".

### Blueprint "Premium":

- Limitación de acceso: Las rutas están limitadas para usuarios del tipo "premium" con una restricción diferente en el número de solicitudes.
- Rutas:
  - /premium/predict: Permite realizar operaciones de predicción para usuarios "premium".
  - /premium/predictById/<predict\_id>: Permite obtener predicciones específicas por ID para usuarios "premium".

## Funciones

Ambos blueprints tienen métodos para GET y POST, requieren autenticación JWT y usan esquemas de validación (PredictSchema y PredictFinishedSchema) para la entrada y salida de datos.

**convertir\_a\_entrada\_modelo(predict\_data):** Esta función recibe datos (predict\_data) de tipo PredictModel y los convierte en un formato adecuado para ser procesados por un modelo neuronal. Transforma los datos a una estructura compatible (un arreglo NumPy en este caso) para la entrada del modelo.

**usar\_modelo\_neuronal(predict\_data):** Utiliza un modelo neuronal para realizar una predicción basada en los datos proporcionados. Accede al modelo neuronal desde la configuración de la aplicación (current\_app.config["modeloNeuronal"]), convierte los datos de entrada usando la función anterior y devuelve la predicción como un número de punto flotante.

**Predict (Clase MethodView)** - Ruta /<tipo\_de\_usuario>/predict:

- **GET:** Requiere autenticación JWT y verifica el tipo de usuario. Si es correcto, devuelve todas las predicciones almacenadas (`PredictModel.query.all()`). Si no, devuelve un error 403 (prohibido).
- **POST:** Requiere autenticación JWT y datos de predicción (`PredictSchema`). Verifica el tipo de usuario. Si es válido, crea una nueva predicción utilizando los datos proporcionados, realiza una predicción utilizando el modelo neuronal, y basado en esa predicción, establece el campo `riesgoCardiaco` en la instancia de la clase `PredictModel`. Luego, agrega esta predicción a la base de datos y la devuelve como respuesta. Si ocurre algún error de base de datos, devuelve un error 400 (bad request). Si el usuario no valida su tipo, devuelve un error 403.
- **PredictById<tipo\_de\_usuario>** (Clase `MethodView`) - Ruta `/<tipo de usuario>/predictById/<predict_id>`:
  - GET: Requiere autenticación JWT y verifica el tipo de usuario. Si es válido, devuelve la predicción con el ID proporcionado (`PredictModel.query.get_or_404(predict_id)`). Si no, devuelve un error 403 (prohibido).

### Anotaciones

- `@blueprint.route()`: Define rutas para el blueprint con sus correspondientes métodos HTTP.
- `@blueprint.response()`: Especifica el esquema de respuesta para las rutas.
- `@blueprint.arguments()`: Define los esquemas de argumentos que se esperan en las solicitudes.
- `@jwt_required(fresh=True)`: Requiere autenticación JWT fresca para acceder a las rutas.

### user.py

Este código implementa varias funcionalidades relacionadas con la gestión de usuarios en una API Flask. Aquí está la explicación de cada parte:

### Blueprint "user":

- Rutas:
  - `/register`: Permite registrar nuevos usuarios en la aplicación. Verifica si el usuario ya existe y, si no, guarda la información del nuevo usuario en la base de datos.
  - `/login`: Permite que los usuarios inicien sesión verificando sus credenciales (nombre de usuario y contraseña) y generando tokens de acceso y actualización JWT.
  - `/SwitchToPremium`: Permite cambiar el tipo de usuario a "premium".
  - `/SwitchToFreemium`: Permite cambiar el tipo de usuario a "freemium".
  - `/refresh`: Permite refrescar el token de acceso JWT.
- Funciones:
  - `UserRegister`: Procesa el registro de un nuevo usuario. Verifica si el usuario ya existe en la base de datos y, si no, guarda la información del nuevo usuario.
  - `UserLogin`: Autentica a los usuarios verificando las credenciales (nombre de usuario y contraseña) y genera tokens de acceso y actualización JWT.

- SwitchToPremium y SwitchToFreemium: Cambian el tipo de usuario de "premium" a "freemium" y viceversa, si las credenciales son válidas.
  - UserRefresh: Refresca el token de acceso JWT para usuarios autenticados.
- Funcionalidades adicionales:
  - Se utilizan esquemas (UserRegisterSchema, UserSchemaBasic) para validar los datos de entrada y salida.
  - Se utiliza passlib para encriptar y verificar contraseñas almacenadas en la base de datos.
  - Se manejan excepciones de SQLAlchemy para posibles errores de base de datos durante las operaciones de escritura.

## Directorio Instance

Dentro está guardada la instancia de la base de datos creada localmente.

## Directorio Migrations

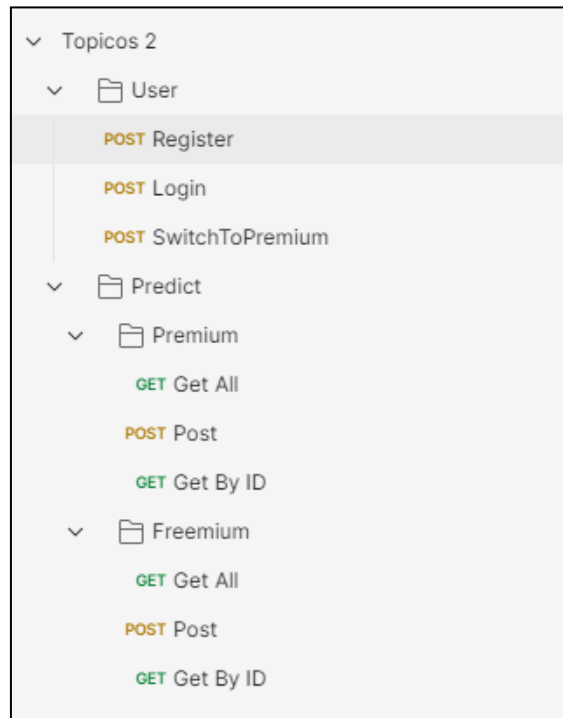
Contiene un archivo env.py, además de muchos más, que entre otras cosas tiene las siguientes funciones para realizar migraciones de la base de datos:

- Configuración de logging: Se configura la información de registro para Alembic utilizando la configuración de logging de Python. Esto permite registrar eventos y mensajes durante las migraciones.
- Obtención del motor de la base de datos: Se definen funciones (get\_engine y get\_engine\_url) para obtener el motor de la base de datos a partir de la aplicación Flask. Esto se hace de manera dinámica para trabajar tanto con Flask-SQLAlchemy anterior como con la versión más reciente.
- Configuración de la URL de SQLAlchemy: Se establece la URL de SQLAlchemy utilizando el motor de la base de datos obtenido previamente.
- Funciones para ejecutar migraciones en modo 'offline' y 'online':
  - run\_migrations\_offline(): Configura el contexto con una URL, no crea un motor, y ejecuta las migraciones sin requerir la presencia de una DBAPI.
  - run\_migrations\_online(): En este modo, se crea un motor y se asocia una conexión al contexto.
- Callbacks para el proceso de revisiones: Se define un callback (process\_revision\_directives) para evitar la generación automática de una nueva migración cuando no hay cambios detectados en el esquema.
- Ejecución de migraciones en modo online o offline: Se verifica el contexto y se ejecuta la función correspondiente según el modo detectado.

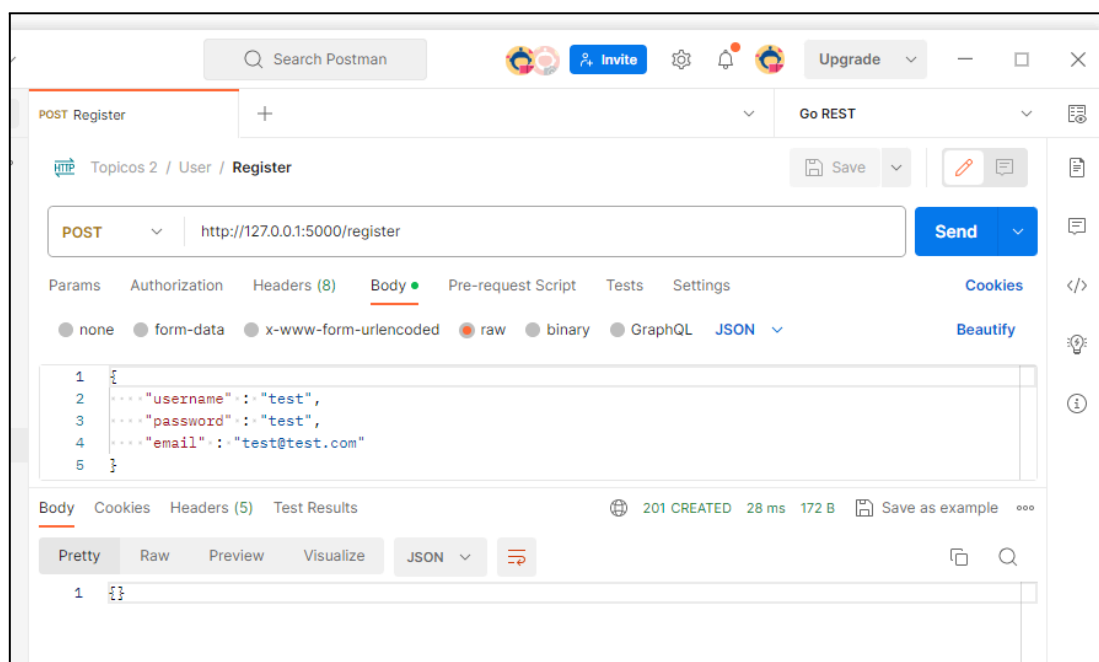


# Pruebas

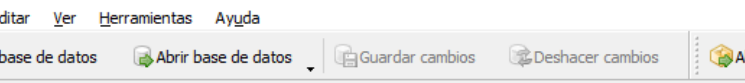
La colección de nuestros servicios para probarlos con Postman se encuentra disponible en la carpeta Servicios, el archivo json se llama **Temas 2.postman\_collection**. Una vez importada la colección se puede ver en Postman los siguientes endpoints:



Como primer paso vamos a registrar un nuevo usuario **test** para hacer nuestras pruebas:



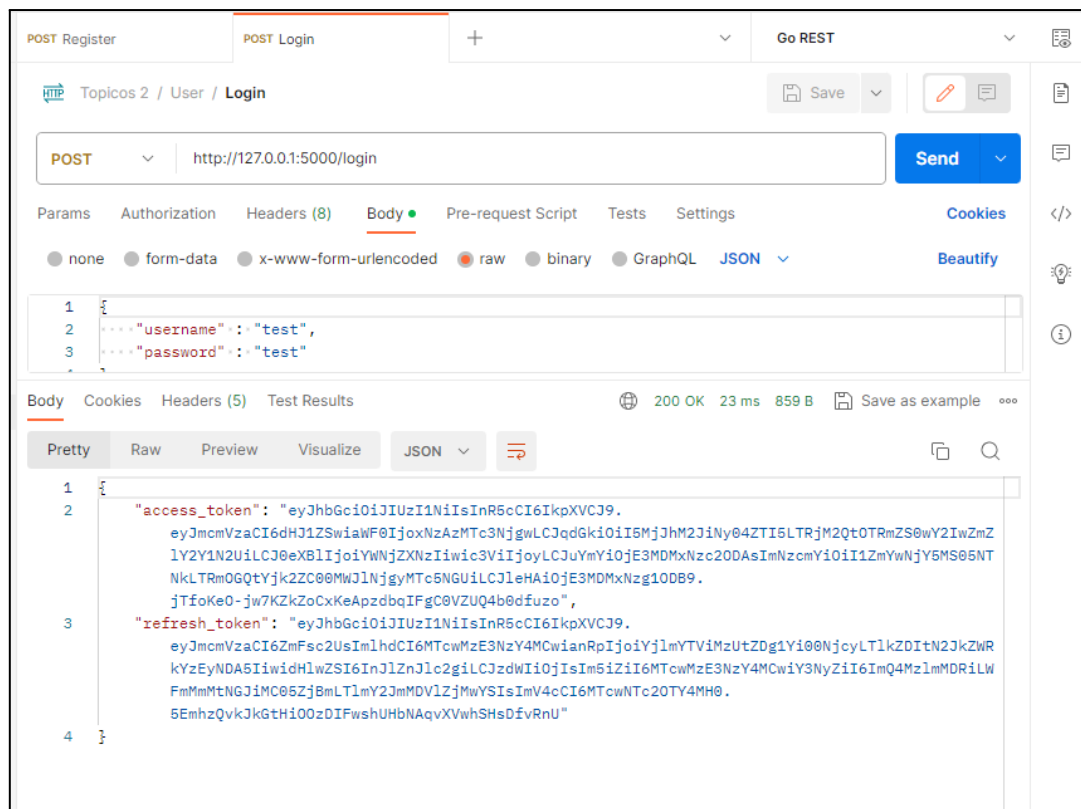
Al revisar nuestra base de datos SQLite3 podemos ver que el usuario se creó correctamente, por defecto los nuevos usuarios son del tipo **freemium**.



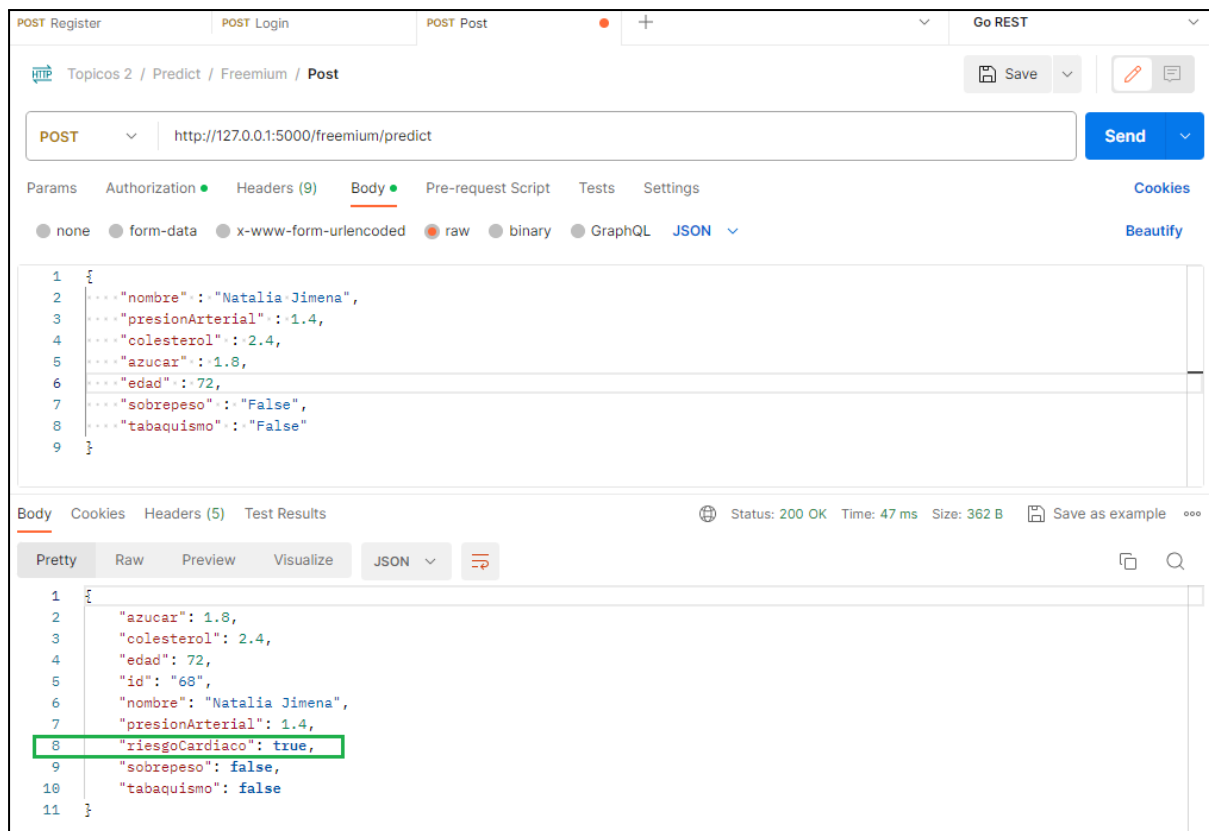
The screenshot shows the DB Browser for SQLite interface. At the top, the title bar reads "DB Browser for SQLite - C:\workspace-spring\tpfinal\_topicos\_2\1\_flask\instance\db.sqlite3". Below the title bar is a menu bar with "Archivo", "Editar", "Ver", "Herramientas", and "Ayuda". The main toolbar contains icons for "Nueva base de datos", "Abrir base de datos", "Guardar cambios", "Deshacer cambios", and "Abrir proyecto". Below the toolbar is a tabbed interface with "Estructura", "Hoja de datos", "Editar pragmas", and "Ejecutar SQL". The "Hoja de datos" tab is active, showing a table named "users" selected from a dropdown menu. The table has five columns: "id", "username", "password", "email", and "type". Each column has a "Filtro" (Filter) button. The table contains two rows of data:

	id	username	password	email	type
1	1	user	\$pbkdf2-...	user@example.com	freemium
2	2	test	\$pbkdf2-...	test@test.com	freemium

Luego nos logueamos con el usuario **test** para obtener los tokens de acceso y actualización JWT, los usuarios **freemium** solo puede hacer 5 solicitudes por minuto.

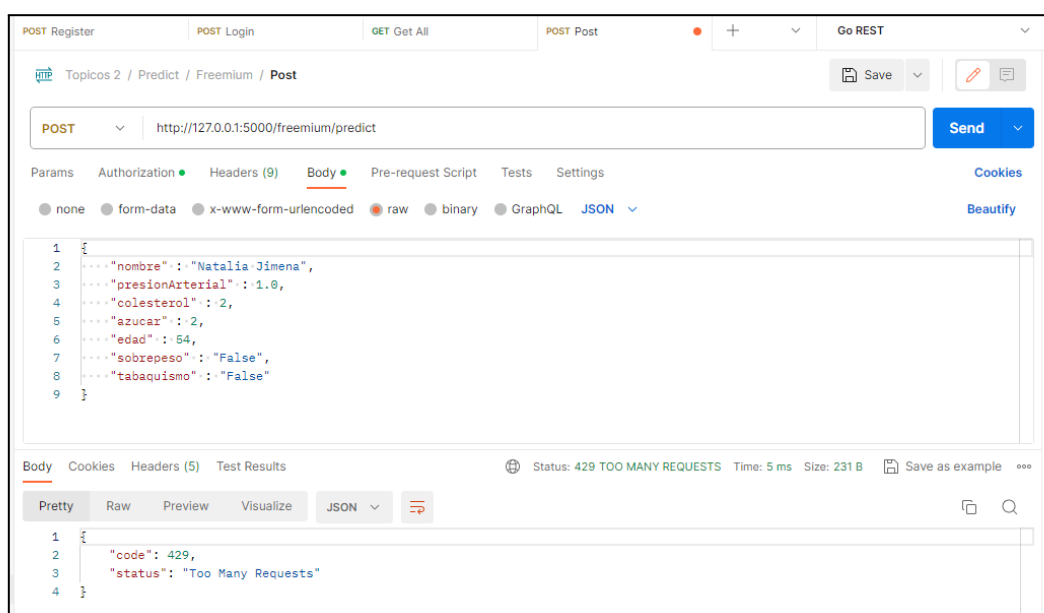


Procedemos a solicitar una predicción con el usuario test usando la información cargada en el **Body** y el token en **Autorization**. La información que cargamos es el segundo registro que [1,2,4,1,4,1,8,72,0,0,1] que encontramos en el archivo **datos\_de\_pacientes\_5000**.



El resultado obtenido **riesgoCardiaco: true** concuerda con el esperado, por lo que el modelo predice correctamente.

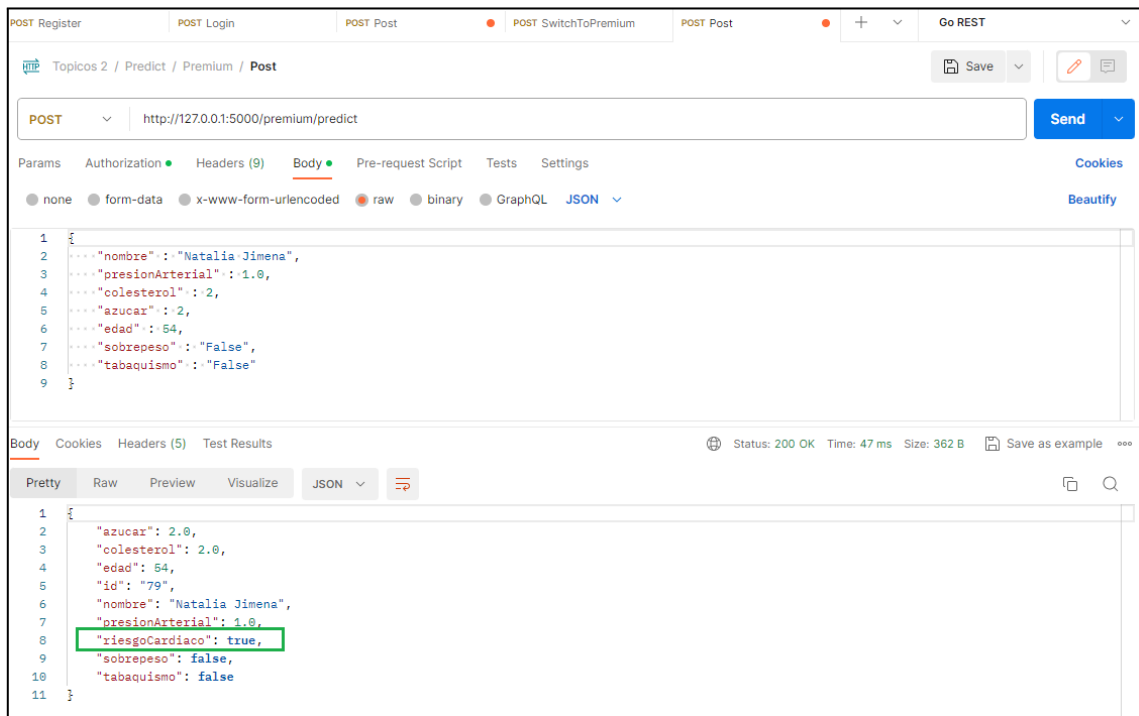
Después de que el usuario test supera su límite de peticiones (5 solicitudes por minuto), obtiene la response **"too many requests"**:



A continuación vamos a pasar el usuario test de freemium a premium:

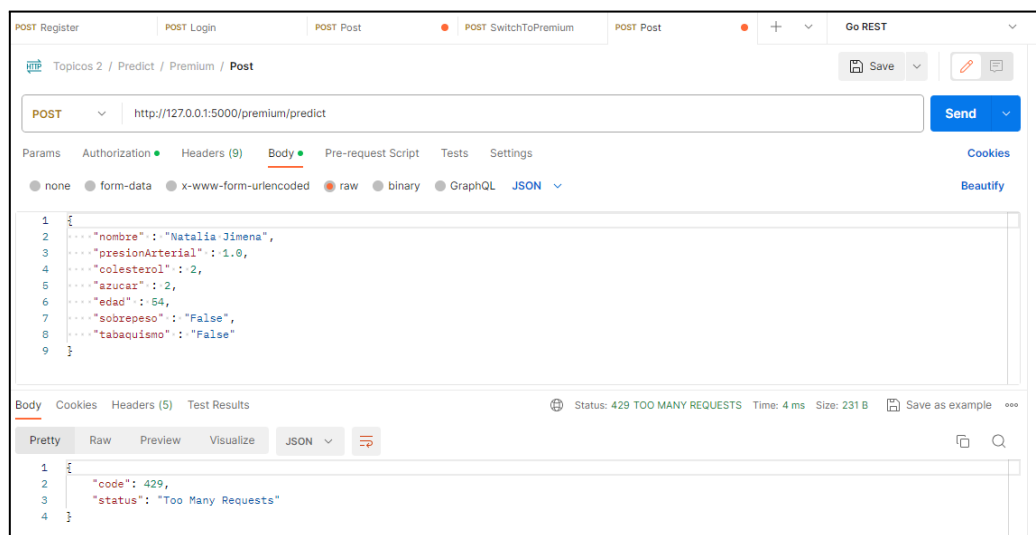


Al realizar una petición con nueva información en el **Body** y con el nuevo token en **Autorization** todo sigue funcionando normalmente:



El resultado obtenido **riesgoCardiaco: true** concuerda con el esperado, por lo que el modelo predice correctamente.

Al superar las 50 solicitudes por minuto de un usuario premium se obtiene la response “**Too Many Requests**”:



# Conclusiones

El presente estudio representa una exhaustiva exploración en la construcción y entrenamiento de un modelo de red neuronal destinado a la predicción del riesgo cardíaco basado en datos clínicos de pacientes.

La arquitectura de la red neuronal secuencial, compuesta por capas con funciones de activación ReLU y una capa de salida con activación Sigmoid, ha mostrado ser efectiva en la clasificación precisa del riesgo cardíaco. La fase de entrenamiento, con 150 épocas y un tamaño de lote de 8, ha permitido alcanzar un modelo con métricas de pérdida y precisión óptimas, evaluadas mediante gráficos que validan el rendimiento del modelo a lo largo del tiempo. Además, la normalización de datos y la segmentación adecuada entre conjuntos de entrenamiento y prueba han asegurado la robustez y generalización del modelo.

La implementación práctica a través de una API en Flask ha facilitado la interacción con este modelo, garantizando la seguridad mediante autenticación JWT, la validación de datos y la gestión eficiente de usuarios.

Nos sirvió como una buena práctica para pulir nuestras habilidades de Flask y al mismo tiempo aprender la aplicación real de un modelo de IA. Nos sentimos muy contentos con el trabajo y las pruebas realizadas, ayudando también a nuestro desarrollo profesional.