

UNIVERSIDAD NACIONAL DE LA PLATA
MAESTRÍA EN INGENIERÍA DE SOFTWARE



Trabajo Practico Final

Asignatura: Topicos de Ingenieria de Software

Año: 2023

Título: Trabajo Integrador: Servicio Web (API)

Autores:

Gomes de Oliveira Neto, Luiz

Mascelloni, Mariela

Docentes:

Prof. Andres Diaz Pace

Prof. Claudia Pons

Prof. Gabriela Perez

Prof. Matias Urbieto

Requisitos Obligatorios para Ejecutar el Proyecto

1. **Acceso al Repositorio:** <https://github.com/engluizgomes/PracticoTOP2>
2. **Python (Versión 3.11):**
 - Asegúrate de tener Python instalado en tu sistema.
 - La versión requerida es la 3.11.6 Puedes descargarla desde python.org.
3. **Visual Studio Code:**
 - Necesitarás tener instalado Visual Studio Code, un entorno de desarrollo de código abierto.
 - Puedes descargarlo desde Visual Studio Code.
4. **Redis:**
 - La aplicación depende de Redis como sistema de almacenamiento en caché.
 - Asegúrate de tener Redis instalado. Puedes encontrar información de instalación en redis.io.
5. **Postman:**
 - Se recomienda tener Postman para probar y documentar las API.
 - Descarga e instala Postman desde getpostman.com.

Pasos para ejecutar exitosamente el Proyecto

1. Ir a Visual Studio Code

2. Ir a la Carpeta del Proyecto de Machine Learning:

Abre la consola de Visual Studio Code y navega a la carpeta del proyecto de machine learning.
Comando: `cd practico\0_ml`.

3. Instalar Dependencias para Machine Learning:

Ejecuta los siguientes comandos para instalar las dependencias necesarias:

```
pip install tensorflow scikit-learn matplotlib
pip install pandas
pip install scikit-learn
```

4. Actualizar Pip:

Para actualizar pip, ejecuta el siguiente comando:

```
python.exe -m pip install --upgrade pip
```

5. Generar Predicción y Guardar el Modelo:

En la consola de Visual Studio Code, ejecuta el siguiente comando para generar la predicción y guardar el modelo:

```
cd practico\0_ML
python .\IA-riesgo.py
```

El modelo se almacenará en `'model.keras'`.

Para ejecutar IA-riesgo.ipynb (Cuaderno de Jupyter):

Si decides ejecutar el cuaderno de Jupyter `'IA-riesgo.ipynb'`, asegúrate de tener instalado `'seaborn'`. Puedes instalarlo con el siguiente comando:

```
pip install seaborn
```

Luego, puedes abrir y ejecutar el cuaderno utilizando tu entorno de Jupyter.

Nota: En este escenario, ten en cuenta que el modelo no se guarda, ya que la funcionalidad de guardar el modelo está asociada específicamente al programa `'IA-riesgo.py'`.

6. Ir a la Carpeta de la Aplicación Flask:

Navega a la carpeta de la aplicación Flask.

Comando: `cd .\1_flask\app.`

7. Crear y Activar el Entorno Virtual:

Crea un entorno virtual con el siguiente comando:

```
python -m venv .venv
```

Activa el entorno virtual:

```
.\venv\Scripts\activate
```

8. Instalación de Dependencias para Flask:

Ejecuta los siguientes comandos para instalar las dependencias en el entorno virtual:

```
python.exe -m pip install --upgrade pip
pip install Flask
pip install --upgrade Flask
pip install redis
pip install numpy
pip install tensorflow
pip install pymongo
pip install requests_cache
```

9. Ejecutar la Aplicación Flask:

Para ejecutar la aplicación Flask, usa el siguiente comando:

```
flask --app flaskr run
```

Decisiones de Diseño

Con respecto al Conjunto de API Keys Válidas:

Actualmente, se utiliza un conjunto predefinido de API keys válidas, por ejemplo, "luiz", "mariela", "mariela2", etc.

Algunas sugerencias y recomendaciones a tener en cuenta para el manejo de API Keys, en un futuro:

- **Para Mayor Seguridad:**

Se podría generar y almacenar dinámicamente API keys utilizando algún método para evitar vulnerabilidades y aumentar la seguridad.

- **Generación Dinámica de API Keys:**

Se podría implementar un método para generar API keys de manera dinámica y segura.

Estas nuevas keys podrían generarse utilizando algún algoritmo criptográfico o un método seguro para garantizar su aleatoriedad y evitar la adivinación.

- **Almacenamiento en la Base de Datos de las mismas:**

La sugerencia es guardar las API keys generadas dinámicamente en una base de datos segura.

Esto proporcionaría una capa adicional de seguridad y facilitaría la gestión de claves en el tiempo.

Con respecto a la definición de la tabla Usuarios:

Por una cuestión de simplificación solo se valida contra la `api_key`, es decir, en Postman se podría haber seteado el usuario y contraseña, además de la `api_key`, y validar los tres elementos (`api_key+usuario+contraseña`).

Funciones Implementadas

- **validarTiempo():**

- **Objetivos:**

La función '**validarTiempo**' tiene como objetivo evaluar diferentes condiciones relacionadas con el tiempo y el usuario para determinar si deben permitirse más consultas.

- **Retornos:**

La función retorna los siguientes códigos:

- 0: Cuando un usuario FREEMIUM ha superado 5 consultas.
 - 1: Cuando un usuario PREMIUM ha superado 50 consultas.
 - 2: Cuando la diferencia entre la primera y la última consulta es mayor a 1 minuto.
 - 8: Cuando no se encuentra el usuario.
 - 9: Actualiza la cantidad de consultas, en un retorno exitoso.

- **Uso de Redis:**

Redis se utiliza para evitar el acceso continuo a la base de datos y mejorar el rendimiento.

- **Tabla hash 'datosCache':**

La información clave, como la API key, el primer tiempo de la consulta, el tipo de usuario y la cantidad de consultas, se guarda en la tabla '**datosCache**'.

Funciones Redis Utilizadas:

- '`cx.lrange("datosCache", 0, -1)`': Devuelve todos los elementos guardados de la tabla hash.
- '`cx.lpush("datosCache", cantCons)`': Inserta un elemento en la tabla hash.
- '`cx.flushall()`': Borra todos los datos de la caché (solo tiene una base de datos aquí).
- '`cx.lset("datosCache", 3, str(cantCons))`': Reemplaza los valores en la tabla hash, en este caso la posición 4 (comienza en 0).

- **predictorFunc():**

Se utiliza el decorador '@lru_cache()' para almacenar los resultados de una función en un caché, sirviendo para optimizar la búsqueda y predicción del modelo, es decir, en aquellos casos en los cuales se invoca la función con los mismos parámetros, devuelve el mismo resultado, sin necesidad recalcularlos .

- **validarParametros():**

Función que valida los parámetros, tanto en cantidad, como que estén dentro de un rango correcto, dependiendo del parámetro.

Funciones Apps Implementadas

- **'auth()'**: Controla la autorización de la API basándose en un conjunto de **'api_key'** posibles.
- **'predictor()'**: Predice el modelo.
- **'ingresar()'**: Ingresa usuarios en la base de datos, no se implementó la modificación de los usuarios, ni la baja de ellos.
- **'logger()'**: Graba en la bitácora (un log).
- **'predict()'**: Realiza la autenticación, obtiene la API Key, valida el tiempo, predice, guarda en la bitácora y formatea el resultado.

Detalle de la función app principal, predict():

1. Autenticación de API Key:

Autentica la API key proporcionada en la solicitud.

2. Obtención de API Key de Postman:

Obtiene la API key de la solicitud de Postman, que puede estar utilizando para realizar solicitudes.

3. Validación de Tiempo y Tipo de Usuario:

Valida el tiempo de la solicitud en relación con el tipo de usuario.

Puede realizar comprobaciones basadas en el tipo de usuario, como limitar el número de consultas.

4. Predicción (predictor):

Utiliza un predictor para realizar una predicción basada en los parámetros proporcionados en la solicitud.

Se asegura de que todos los parámetros necesarios estén presentes y dentro del rango correcto.

5. Registro en Bitácora:

Guarda la información relevante en la bitácora.

Puede incluir detalles como la API key, parámetros de la solicitud, resultados de la predicción, etc.

6. Formateo del Resultado:

Formatea el resultado de la predicción para devolverlo de manera clara y legible en la respuesta.

Aclaración: Se asegura de que todos los parámetros necesarios estén presentes y dentro del rango correcto al realizar la predicción.

Ejemplos de cómo se convocan a las funciones apps

1. Predicción:

<http://127.0.0.1:5000/predict?colesterol=2.4&presion=1.4&glucosa=1.8&edad=72&sobrepeso=0&tabaquismo=0>

2. Ingreso de Usuario:

<http://127.0.0.1:5000/ingresar?usuario=mariela&contraseña=mariela&tipo=PREMIUM>

3. Registro en Bitácora:

<http://127.0.0.1:5000/logger?colesterol=2.4&presion=1.4&glucosa=1.8&edad=72&sobrepeso=0&tabaquismo=0>

4. Autenticación:

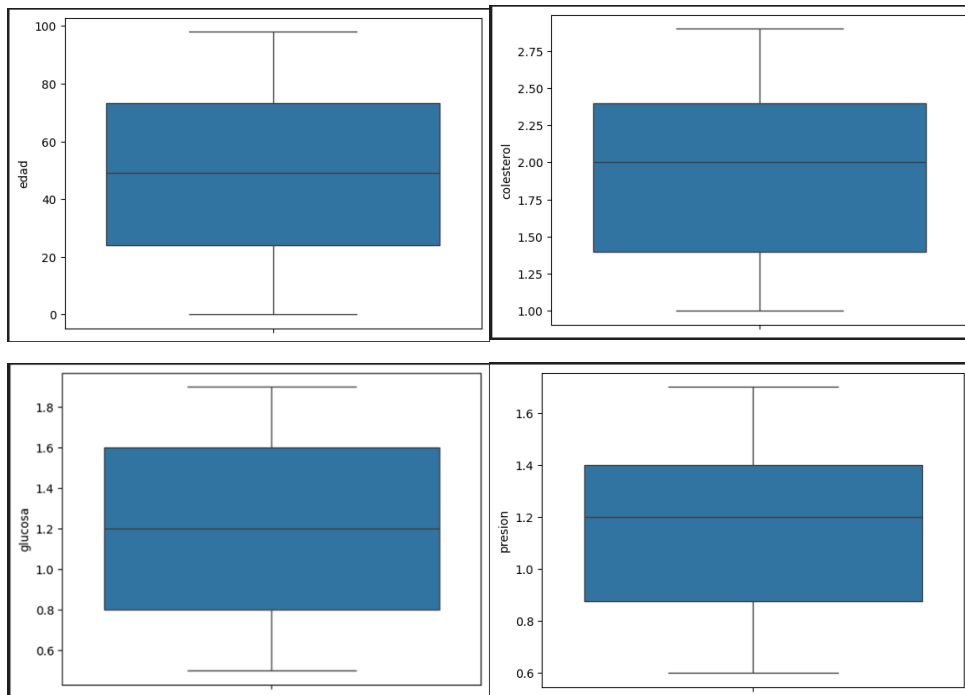
<http://127.0.0.1:5000/auth>

5. Predicción con Caché:

<http://127.0.0.1:5000/predictor?colesterol=2.4&presion=1.4&glucosa=1.8&edad=72&sobrepeso=0&tabaquismo=0>

Análisis de Datos con IA

Tenemos que los parámetros oscilan según las siguientes graficas:



División del Conjunto de Datos:

Después de cargar el conjunto de datos, se dividió en conjuntos de entrenamiento y prueba. Esta práctica es fundamental para evaluar el rendimiento del modelo en datos no vistos. Hemos dividido el conjunto de datos en conjuntos de entrenamiento y prueba utilizando la función `train_test_split` de `scikit-learn`. Esta división es esencial para evaluar el rendimiento del modelo en datos no observados.

- Separar los datos de entrada X y los datos de salida Y

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

Escalado de Variables Numéricas:

Para garantizar una convergencia eficiente y la estabilidad del modelo, escalamos las variables numéricas utilizando la clase **StandardScaler** de **scikit-learn**. Este paso es crucial para algoritmos sensibles a la escala, como la regresión logística. Escalamos los conjuntos de entrenamiento y prueba por separado para evitar fugas de información entre ellos.

- Escalar variables numéricas para el conjunto de entrenamiento

```
scaled_X_train = scaler.fit_transform(X_train)
scaled_X_train = pd.DataFrame(scaled_X_train, columns=X_train.columns)
```

- Escalar variables numéricas para el conjunto de prueba

```
scaled_X_test = scaler.fit_transform(X_test)
scaled_X_test = pd.DataFrame(scaled_X_test, columns=X_test.columns)
```

Conjunto de Entrenamiento Escalado:

Después de escalar las variables numéricas para el conjunto de entrenamiento, convertimos los datos escalados de nuevo a un **DataFrame** de Pandas. Esto nos permite visualizar las primeras filas del conjunto de entrenamiento escalado, proporcionando una comprensión inicial de la transformación aplicada.

- Mostrar las primeras filas de los conjuntos escalados para el conjunto de entrenamiento

```
print("Conjunto de Entrenamiento Escalado:")
print(scaled_X_train.head())
```

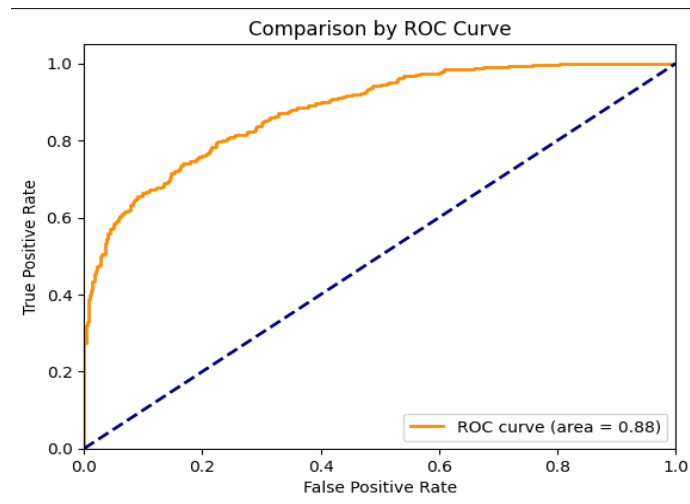
Conjunto de Prueba Escalado:

De manera similar al conjunto de entrenamiento, escalamos las variables numéricas en el conjunto de prueba utilizando el mismo escalador ajustado al conjunto de entrenamiento. Los datos escalados también se convierten en un **DataFrame** de Pandas para facilitar la inspección de las primeras filas y garantizar consistencia en los procedimientos aplicados.

- Mostrar las primeras filas de los conjuntos escalados para el conjunto de prueba

```
print("\nConjunto de Prueba Escalado:")
print(scaled_X_test.head())
```

Se puede observar que la regresión lógica del modelo utilizado es:



A medida que entrenamos el modelo, se obtienen diversas precisiones durante la evaluación. Por ejemplo, al realizar la evaluación del modelo con la siguiente instrucción:

```
score = model.evaluate(X_test, y_test, verbose=0)
print('Precisión:', score[1])
```

Podemos observar que las precisiones varían, obteniendo valores como **0.7639999985694885**, **0.7789999842643738**, ..., **0.875**. Estos valores representan la precisión del modelo en el conjunto de prueba en diferentes momentos durante el entrenamiento. La variación en estos números puede ser indicativa de cómo el modelo está aprendiendo y generalizando a lo largo del proceso de entrenamiento. Un aumento en la precisión generalmente sugiere una mejora en el rendimiento del modelo.

En nuestro modelo grabado, que tiene una precisión del **0.921999990940094**, se ha determinado que una persona de 82 años con sobrepeso tiene un riesgo cardiaco del **98,56%**. Este resultado sugiere un alto riesgo cardiovascular para una persona con esas características según las predicciones de nuestro modelo. La precisión del **92.2%** indica la confianza del modelo en sus predicciones, aunque siempre es importante considerar otras variables y fuentes de información al interpretar estos resultados.

<http://127.0.0.1:5000/predict?colesterol=1.4&presion=1.5&glucosa=1.8&edad=82&sobrepeso=1&tabaquismo=0...>
Save

POST
http://127.0.0.1:5000/predict?colesterol=1.4&presion=1.5&glucosa=1.8&edad=82&sobrepeso=1&tabaquismo=0...
Send

Params
Authorization
Headers (9)
Body
Pre-request Script
Tests
Settings
Cookies

Headers
8 hidden

	Key	Value	Description	...	Bulk Edit	Presets
<input checked="" type="checkbox"/>	Authorization	mariela2				
	Key	Value	Description			

body
Cookies
Headers (5)
Test Results
200 OK
1137 ms
202 B
Save as example

Pretty
Raw
Preview
Visualize
HTML

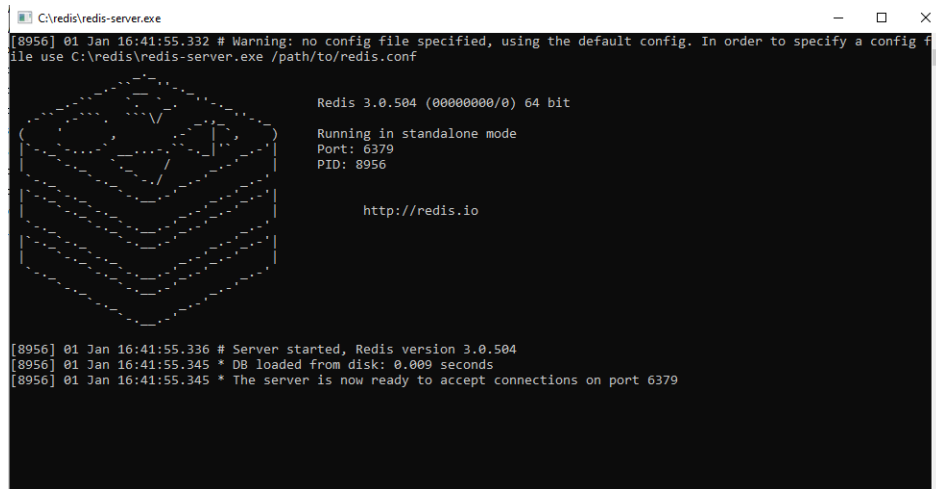
1 Tiene un riesgo de 98.56147 %

Programas Python del Proyecto

- **'IA-riesgo.py'**: Realiza la inteligencia para calcular el riesgo cardíaco y guarda el modelo.
- **'_init_.py'**: Define las funciones más importantes.
- **'db.py'**: Define el acceso a la base de datos.

Para la ejecución:

- 1) Se debe ejecutar **redis-server**



```
C:\redis\redis-server.exe
[8956] 01 Jan 16:41:55.332 # Warning: no config file specified, using the default config. In order to specify a config file use C:\redis\redis-server.exe /path/to/redis.conf

Redis 3.0.504 (00000000/0) 64 bit

Running in standalone mode
Port: 6379
PID: 8956

http://redis.io

[8956] 01 Jan 16:41:55.336 # Server started, Redis version 3.0.504
[8956] 01 Jan 16:41:55.345 * DB loaded from disk: 0.009 seconds
[8956] 01 Jan 16:41:55.345 * The server is now ready to accept connections on port 6379
```

- 2) Se debe ejecutar **redis-cli**



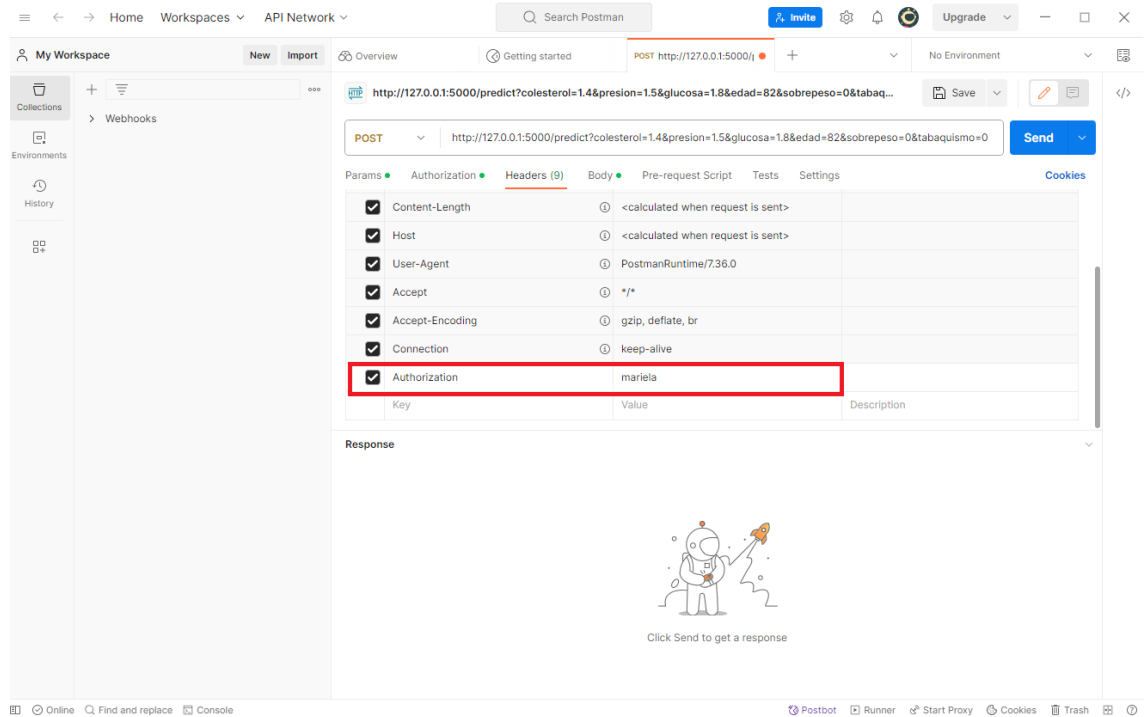
```
C:\redis\redis-cli.exe
127.0.0.1:6379>
```

- 3) En la consola, ejecuta el script de la aplicación. Escribe:

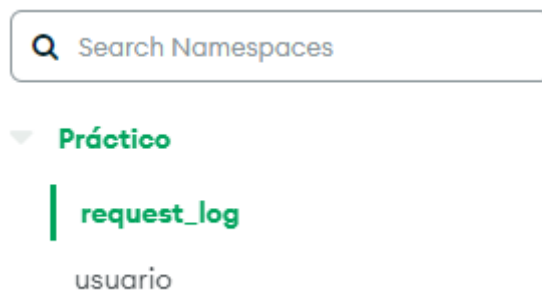
```
Mode                LastWriteTime         Length Name
-----
d-----            1/1/2024      15:57           .venv
d-----            24/12/2023     14:06         flaskr

PS C:\practico\1_flask\app> flask --app flaskr run
```

- 4) Abre Postman, la herramienta que utilizarás para realizar solicitudes a tu aplicación. En la sección de encabezados (Headers), ingresa en el campo Authorization una de las siguientes claves: 'mariela', 'mariela2' o 'luiz'.



- 5) Teniendo la siguiente Base de Datos Práctico:



Con usuario y contraseña, como sigue:

Usuario: mmascelloni@yahoo.com.ar
Contraseña: Mariela2023

Con los siguientes usuarios cargados:

```
_identificación: Id. de objeto(' 65899e3aed7e9722f2ebb501 ')  
Clave : " mariela "  
API  
usuario: " mariela "  
contraseña: " mariela "  
tipo: " DE PRIMERA CALIDAD "
```

```
_identificación: Id. de objeto (' 6589a01fed7e9722f2ebb511 ')  
Clave : " mariela2 "  
API  
usuario: " mariela "  
contraseña: " mariela "  
tipo: " GRATIS "
```

```
_identificación: Id. de objeto(' 658b3fbdbe4f95854eb5f59c ')  
Clave : " mariela24 "  
API  
usuario: " mariela3 "  
contraseña: " mariela "  
tipo: " DE PRIMERA CALIDAD "
```

```
_identificación: Id. de objeto(' 658b413bbe4f95854eb5f5a2 ')  
Clave : " luiz "  
API  
usuario: " luiz "  
contraseña: " luiz "  
tipo: " GRATIS "
```

- Y una bitácora (log), donde se registra cada consulta exitosa, dejando un historial de las mismas.

Práctico.request_log

STORAGE SIZE: 44KB LOGICAL DATA SIZE: 22.76KB TOTAL DOCUMENTS: 127 INDEXES TOTAL SIZE: 36KB

[Find](#) [Indexes](#) [Schema Anti-Patterns](#) ⓘ [Aggregation](#) [Search Indexes](#)

[Filter](#) ⓘ Type a query: { field: 'value' }

QUERY RESULTS: 1-20 OF MANY

```

_id: ObjectId('65899ed5ed7e9722f2ebb504')
timestamp: "2023-12-25T12:25:09.033864"
▶ params: Array (7)
  response: "[[0.64441836]]"

```

```

_id: ObjectId('65899f57ed7e9722f2ebb506')
timestamp: "2023-12-25T12:27:19.089398"
▶ params: Array (7)
  response: "[[0.64441836]]"

```

```

_id: ObjectId('65899f60ed7e9722f2ebb508')
timestamp: "2023-12-25T12:27:28.563717"
▶ params: Array (7)
  response: "[[0.64441836]]"

```

6) Con éstas **Api_keys** válidas

```

api_keys = {
    "mariela", "mariela2", "mariela3", "luiz"
}

```

Para ejecutar: 'flask --app flaskr run' (en la consola).

Evidencia de prueba

Caso 1: Con un 'api_key= mariela', con usuario **PREMIUM** (existe en la base).

```

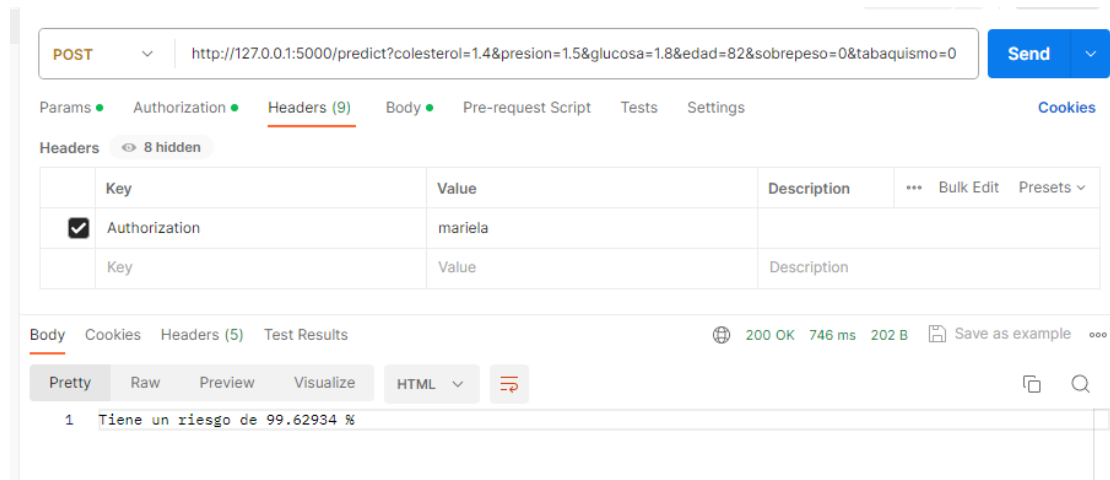
_id: ObjectId('65899e3aed7e9722f2ebb501')
api_key: "mariela"
usuario: "mariela"
contraseña: "mariela"
tipo: "PREMIUM"

```

Consulta:

<http://127.0.0.1:5000/predict?colesterol=1.4&presion=1.5&glucosa=1.8&edad=82&sobrepeso=0&tabaquismo=0>

Postman devuelve:

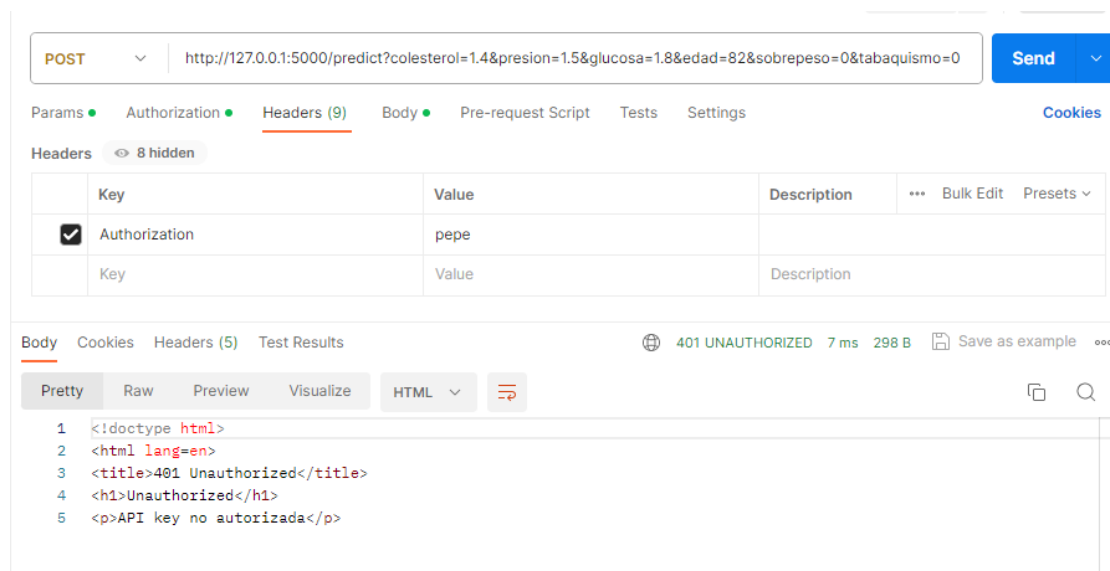


Caso 2: Con `api_key` no válida, es decir, que no está en el conjunto válido de `api_key` ("pepe").

Consulta:

<http://127.0.0.1:5000/predict?colesterol=1.4&presion=1.5&glucosa=1.8&edad=82&sobrepeso=0&tabaquismo=0>

Postman devuelve:



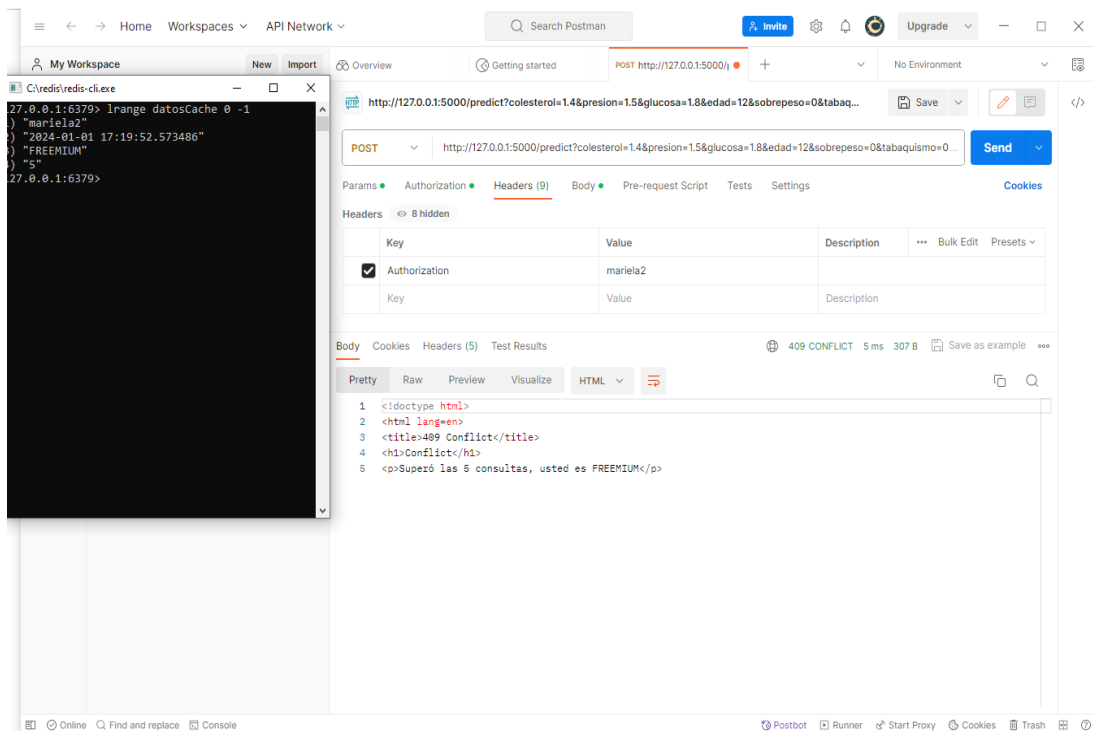
Caso 3: Con `api_key` dentro del conjunto válido de `api_keys` ("mariela2") y con el usuario registrado en la base (usuario de tipo **FREEMIUM**).

Cuando solicita más de 5 consultas, dentro de un minuto.

Consulta:

<http://127.0.0.1:5000/predict?colesterol=1.4&presion=1.5&glucosa=1.8&edad=12&sobrepeso=0&tabaquismo=0>.

Postman devuelve:



Caso 4: con `api_key` que está en el conjunto válido de `api_key` ("mariela3"), la `api_key` del usuario no está en la base.

Consulta:

<http://127.0.0.1:5000/predict?colesterol=2.4&presion=1.4&glucosa=1.8&edad=72&sobrepeso=0&tabaquismo=0>

Postman devuelve:

POST ▼ http://127.0.0.1:5000/predict?colesterol=2.4&presion=1.4&glucosa=1.8&edad=72&sobrepeso=0&tabaquismo=0 Send ▼

Params ● Authorization ● Headers (9) Body ● Pre-request Script Tests Settings Cookies

Headers 8 hidden

	Key	Value	Description	...	Bulk Edit	Presets
<input checked="" type="checkbox"/>	Authorization	mariela3				
	Key	Value	Description			

Body Cookies Headers (5) Test Results 409 CONFLICT 942 ms 283 B Save as example ...

Pretty Raw Preview Visualize HTML ▼ ≡

```

1 <!doctype html>
2 <html lang=en>
3 <title>409 Conflict</title>
4 <h1>Conflict</h1>
5 <p>No esta el usuario</p>

```

Caso de prueba: creación de usuarios

Consulta:

<http://127.0.0.1:5000/ingresar?usuario=mariela&contraseña=mariela&tipo=PREMIUM>

Postman devuelve:

PUT ▼ http://127.0.0.1:5000/ingresar?usuario=mariela&contraseña=mariela&tipo=PREMIUM Send ▼

Params ● Authorization ● Headers (9) Body ● Pre-request Script Tests Settings Cookies

Headers 8 hidden

	Key	Value	Description	...	Bulk Edit	Presets
<input checked="" type="checkbox"/>	Authorization	mariela2				
	Key	Value	Description			

Body Cookies Headers (5) Test Results 200 OK 885 ms 196 B Save as example ...

Pretty Raw Preview Visualize HTML ▼ ≡

```

1 Ya se cargó el usuario

```

Caso de prueba: logger()

Consulta:

<http://127.0.0.1:5000/logger?colesterol=2.4&presion=1.4&glucosa=1.8&edad=72&sobrepeso=0&tabaquismo=0>

Postman devuelve:

POST ▼ | http://127.0.0.1:5000/logger?colesterol=2.4&presion=1.4&glucosa=1.8&edad=72&sobrepeso=0&tabaquismo=0 Send ▼

Params ● Authorization ● **Headers (9)** ● Body ● Pre-request Script Tests Settings Cookies

Headers 8 hidden

	Key	Value	Description	...	Bulk Edit	Presets ▼
<input checked="" type="checkbox"/>	Authorization	mariela3				
	Key	Value	Description			

Body Cookies Headers (5) Test Results 200 OK 1115 ms 193 B Save as example ...

Pretty Raw Preview Visualize HTML ▼ ≡

```
1 guardé en bitácora
```

Caso de prueba: auth()

1) Api_key válida:

Consulta:

<http://127.0.0.1:5000/auth>

Postman devuelve:

POST ▼ | http://127.0.0.1:5000/auth Send ▼

Params ● Authorization ● **Headers (9)** ● Body ● Pre-request Script Tests Settings Cookies

Headers 8 hidden

	Key	Value	Description	...	Bulk Edit	Presets ▼
<input checked="" type="checkbox"/>	Authorization	mariela3				
	Key	Value	Description			

Body Cookies Headers (5) Test Results 200 OK 4 ms 191 B Save as example ...

Pretty Raw Preview Visualize HTML ▼ ≡

```
1 API key autorizada
```

2) Api_key no válida:

Consulta:

<http://127.0.0.1:5000/auth>

Postman devuelve:

POST http://127.0.0.1:5000/auth Send

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies

Headers 8 hidden

	Key	Value	Description	...	Bulk Edit	Presets
<input checked="" type="checkbox"/>	Authorization	mariela53				
	Key	Value	Description			

body Cookies Headers (5) Test Results 401 UNAUTHORIZED 7 ms 298 B Save as example

Pretty Raw Preview Visualize HTML

```

1 <!doctype html>
2 <html lang=en>
3 <title>401 Unauthorized</title>
4 <h1>Unauthorized</h1>
5 <p>API key no autorizada</p>

```

Caso de prueba: predictor()

Consulta:

<http://127.0.0.1:5000/predictor?colesterol=2.4&presion=1.4&glucosa=1.8&edad=72&sobrepeso=0&tabaquismo=0>

Postman devuelve:

POST http://127.0.0.1:5000/predictor?colesterol=2.4&presion=1.4&glucosa=1.8&edad=72&sobrepeso=0&tabaquismo=0 Send

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies

Headers 8 hidden

	Key	Value	Description	...	Bulk Edit	Presets
<input checked="" type="checkbox"/>	Authorization	mariela53				
	Key	Value	Description			

body Cookies Headers (5) Test Results 200 OK 223 ms 186 B Save as example

Pretty Raw Preview Visualize HTML

```

1 [[0.9712931]]

```