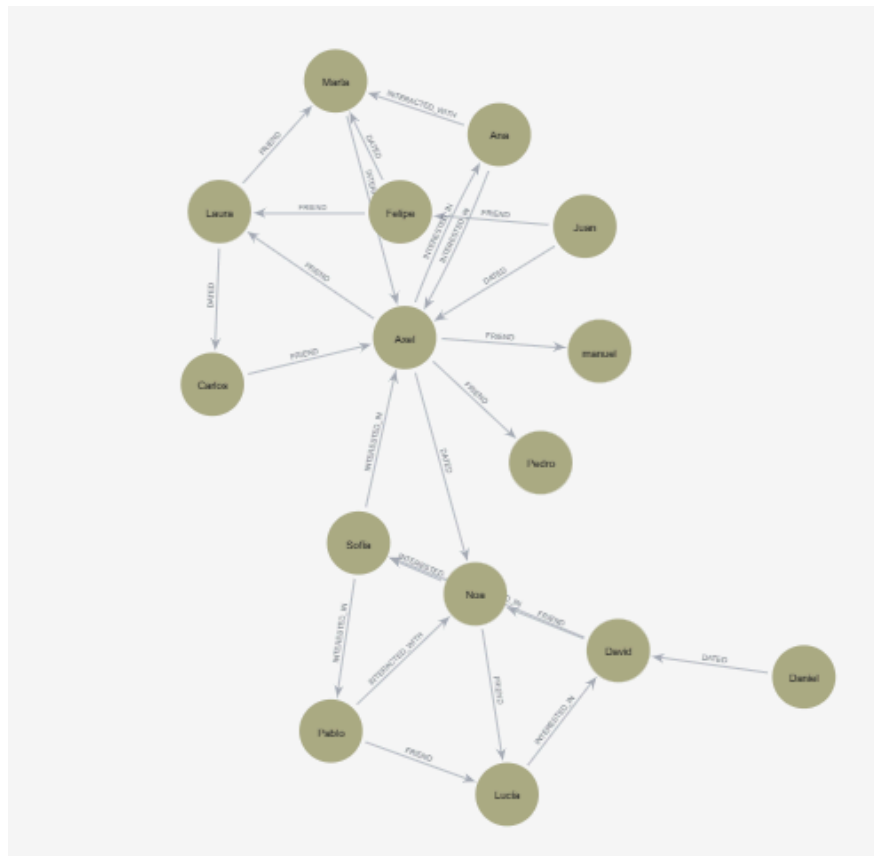


# LOVELINK : LA RED SOCIAL DE RELACIONES



**Axel Berral**

## Iker Infantes

# ÍNDICE

<b>ÍNDICE.....</b>	<b>2</b>
<b>INTRODUCCIÓN.....</b>	<b>3</b>
<b>FASE 1: Entorno y base de proyecto.....</b>	<b>4</b>
1. Crear el entorno virtual.....	4
2. Instalar dependencias iniciales.....	4
3. Crear estructura básica de carpetas.....	4
4. Guardar las dependencias instaladas.....	4
<b>FASE 2: Conexión con base de datos (Neo4j).....</b>	<b>6</b>
1. Instalar y/o configurar Neo4j localmente o en la nube (AuraDB).....	6
2. Crear archivo .env con las credenciales de Neo4j.....	6
3. Crear archivo database.py para manejar la conexión.....	7
4. Probar una consulta de ejemplo desde FastAPI hacia Neo4j.....	7
<b>FASE 3: Modelado del Grafo.....</b>	<b>9</b>
1. Definir nodos Person y relaciones (FRIEND,DATED,INTERACTED_WITH).....	9
NODES: Person.....	9
RELATIONS.....	9
2. Crear un script para poblar la base con algunos datos de prueba ( usando Cypher)..	10
3. Crear funciones para insertar personas y relaciones desde la API.....	11
<b>FASE 4: Lógica de Recomendación.....</b>	<b>16</b>
1. Diseñar y probar algoritmo “friend of a friend con twist romántico”.....	16
2. Añadir filtros opcionales: edad, género, intereses.....	18
<b>FASE 5: Cliente web sencillo.....</b>	<b>24</b>
1. Reestructura mínima del proyecto.....	24
2. Añadir soporte para HTML en FastAPI.....	24
3. Creamos el archivo script.js en la carpeta static:.....	25
<b>FASE 6: Autenticación y usuarios.....</b>	<b>29</b>
1. Registro.....	29
2. Login.....	31
3. Views de Register y Login.....	33
<b>FASE 7: RED SOCIAL FUNCIONAL.....</b>	<b>38</b>
1. Implementar las funciones para los usuarios logueados.....	38
RECOMMENDATIONS.....	38
PATH-TO-PERSON.....	40
PERFIL Y EDITAR TU PERFIL.....	42
AÑADIR FOTOS.....	45
<b>FASE 8: Despliegue en la nube y en internet.....</b>	<b>49</b>
1. Base de datos en la nube.....	49
2. Despliegue en Internet.....	49
<b>Estructura Final del Proyecto.....</b>	<b>50</b>

# INTRODUCCIÓN

Este proyecto consiste en el desarrollo de una API de recomendación de parejas basada en conexiones indirectas, utilizando tecnologías modernas para analizar y extraer valor de las relaciones sociales pasadas. Se trata de una API REST respaldada por una base de datos de grafos (Neo4j) y acompañada de un cliente web sencillo para la interacción del usuario.

## Objetivo del sistema

El objetivo es encontrar "matches" románticos potenciales no evidentes, a partir del análisis de conexiones sociales indirectas: exparejas, amistades, interacciones previas, etc. Inspirado en el algoritmo "Friend of a Friend", el sistema añade un enfoque romántico que prioriza conexiones prometedoras dentro del grafo social.

## Tecnologías utilizadas

- FastAPI para la construcción de la API REST.
- Neo4j como base de datos orientada a grafos.
- Cliente web para visualizar y probar el sistema.

## Conceptos clave

- Inferencia basada en grafos sociales.
- Recomendación de contactos potenciales.
- Respeto por la privacidad y los datos personales.
- Arquitectura modular basada en servicios (API + frontend).

## Metodología

Para llevar a cabo este proyecto de forma estructurada y escalable, lo hemos dividido en 8 fases distintas, que abarcan desde la concepción inicial hasta el despliegue final en la nube.

# FASE 1: Entorno y base de proyecto

## 1. Crear el entorno virtual

```
python3 -m venv venv
```

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

```
python3 -m venv venv
```

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

## 2. Instalar dependencias iniciales

Vamos a instalar **FastAPI**, el servidor **Uvicorn**, y la librería de **Neo4j**:

```
pip install fastapi[all] uvicorn neo4j python-dotenv
```

```
pip install fastapi[all] uvicorn neo4j python-dotenv
```

## 3. Crear estructura básica de carpetas

```
LoveLink/  
├── main.py  
├── database.py  
├── .env  
└── requirements.txt
```

## 4. Guardar las dependencias instaladas

```
pip freeze > requirements.txt
```

```
pip freeze > requirements.txt
```

## 4. Probar que funciona

Hacemos prueba de código en [main.py](#):

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def read_root():
    return {"message": "Hola món! Benvingut a LoveLink 🍷"}
```

lo ejecutamos y ponemos en la terminal este comando:

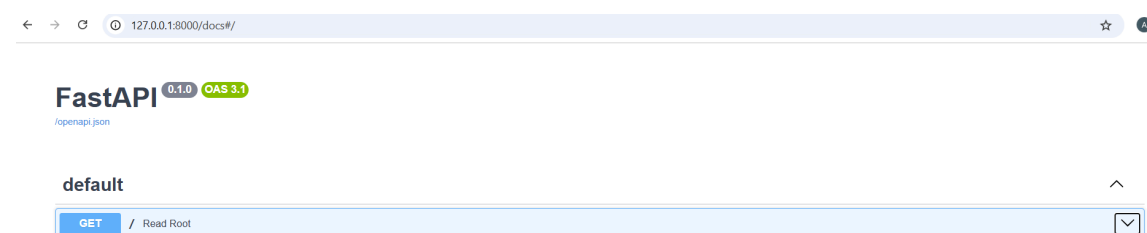
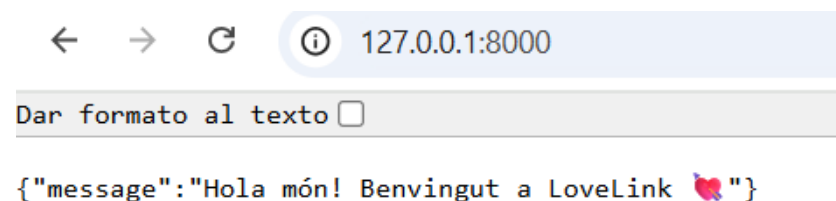
```
uvicorn main:app --reload
```

y comprobamos que funciona:

<http://127.0.0.1:8000>

<http://127.0.0.1:8000/docs/>

```
python.exe C:\Users\Axel\Desktop\Computacio Distribuida i Aplicacions\PR2\LoveLink\main.py
PS C:\Users\Axel\Desktop\Computacio Distribuida i Aplicacions\PR2\LoveLink> uvicorn main:app --reload
INFO: Will watch for changes in these directories: ['C:\Users\Axel\Desktop\Computacio Distribuida i Aplicacions\PR2\LoveLink']
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process [34980] using WatchFiles
INFO: Started server process [5076]
INFO: Waiting for application startup.
INFO: Application startup complete.
INFO: 127.0.0.1:50448 - "GET / HTTP/1.1" 200 OK
```



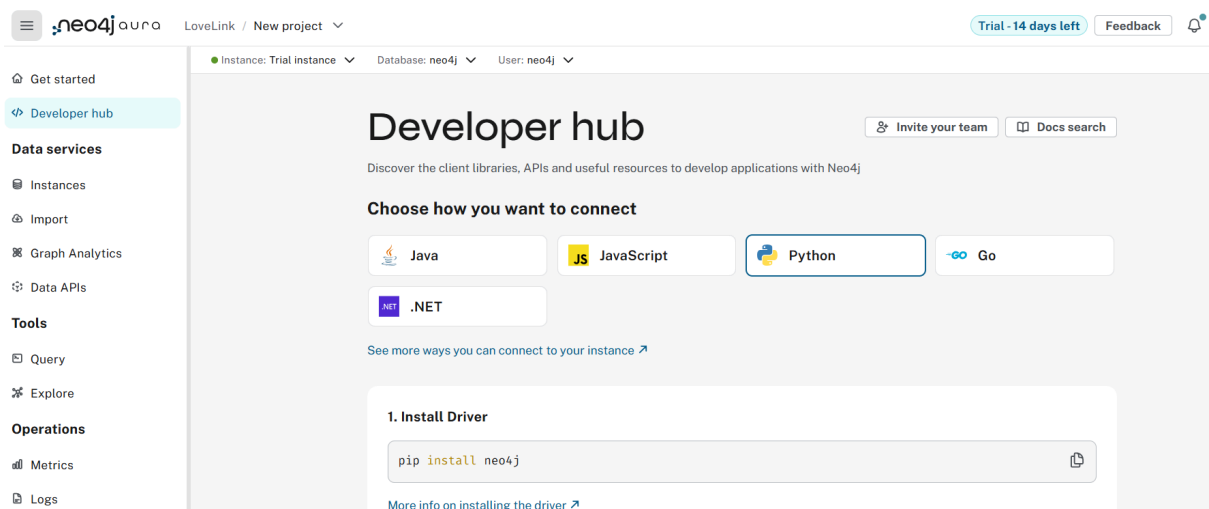
## FASE 2: Conexión con base de datos (Neo4j)

### 1. Instalar y/o configurar Neo4j localmente o en la nube (AuraDB)

Crea una cuenta gratuita en <https://neo4j.com/cloud/aura/>

Elige la opción "**Aura Free**".

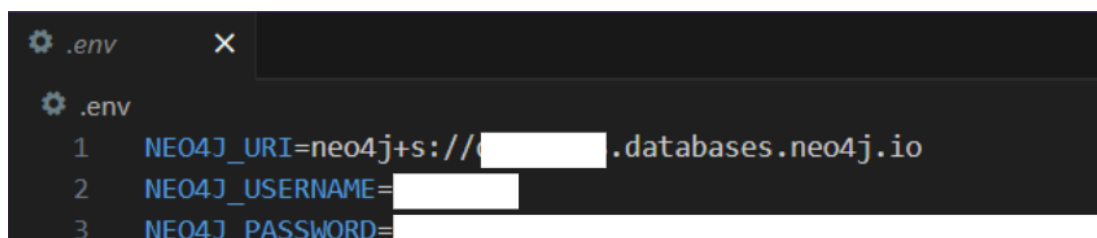
Crea una base de datos gratuita (tiene un límite, pero es más que suficiente).



1. Apuntamos los siguientes datos:

- **URI (bolt)** → `neo4j+s://vuestra uri.databases.neo4j.io`
- **Usuario** → `neo4j`
- **Contraseña** → `vuestra contraseña`

### 2. Crear archivo `.env` con las credenciales de Neo4j



### 3. Crear archivo `database.py` para manejar la conexión

Añadimos esto en el [database.py](#):

```
database.py M X
database.py > ...
1  from neo4j import GraphDatabase
2  from dotenv import load_dotenv
3  import os
4
5  # Cargar variables del archivo .env
6  load_dotenv()
7
8  NEO4J_URI = os.getenv("NEO4J_URI")
9  NEO4J_USERNAME = os.getenv("NEO4J_USERNAME")
10 NEO4J_PASSWORD = os.getenv("NEO4J_PASSWORD")
11
12 # Crear el driver de conexión
13 driver = GraphDatabase.driver(
14     NEO4J_URI,
15     auth=(NEO4J_USERNAME, NEO4J_PASSWORD)
16 )
17
18 # Función de utilidad para ejecutar consultas
19 def run_query(query, parameters=None):
20     with driver.session() as session:
21         result = session.run(query, parameters or {})
22         return [record.data() for record in result]
23
```

### 4. Probar una consulta de ejemplo desde FastAPI hacia Neo4j

Modificamos el [Main.py](#):

```
main.py M X
main.py > test_connection
1  from fastapi import FastAPI
2  from database import run_query
3
4  app = FastAPI()
5
6  @app.get("/")
7  def read_root():
8      return {"message": "Hola món! Benvingut a Lovelink 🍷"}
9
10 @app.get("/test-neo4j")
11 def test_connection():
12     query = "MATCH (n) RETURN n LIMIT 5"
13     try:
14         results = run_query(query)
15         return {"success": True, "data": results}
16     except Exception as e:
17         return [{"success": False, "error": str(e)}]
```

Ejecutamos de nuevo:

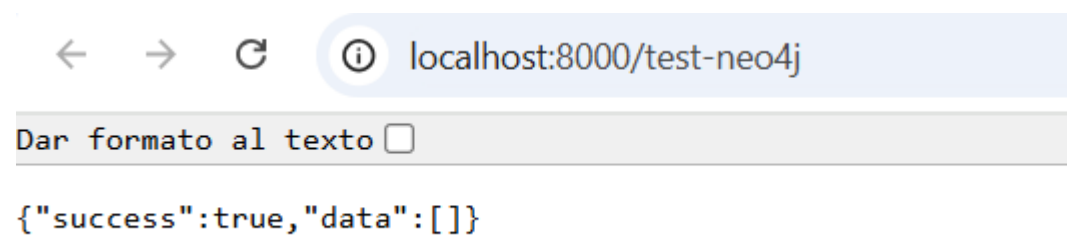
uvicorn main:app --reload

```
PS C:\Users\Axel\Desktop\Computacio Distribuida i Aplicacions\PR2\LoveLink> uvicorn main:app --reload
INFO: Will watch for changes in these directories: ['C:\\Users\\Axel\\Desktop\\Computacio Distribuida i
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process [4104] using WatchFiles
INFO: Started server process [33284]
INFO: Waiting for application startup.
INFO: Application startup complete.
```

y vamos a :

<http://localhost:8000/test-neo4j>

y comprobamos que funciona.



#### Trial instance

● RUNNING

ID: dfd90603

Connect ⓘ ⋮

Type: AuraDB Professional Memory: 2GB CPU: 1 Storage: 4GB AWS / Europe, Paris (eu-west-3)

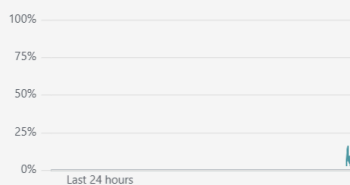
14 days trial remaining

Configure

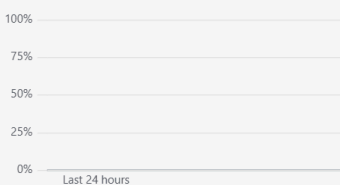
#### Metrics

↻ View all metrics

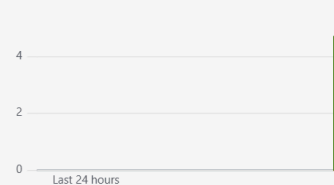
CPU Usage



Storage



Query Rate



```
INFO: 127.0.0.1:63230 - "GET /test-neo4j HTTP/1.1" 200 OK
```



## FASE 3: Modelado del Grafo

### 1. Definir nodos Person y relaciones (FRIEND,DATED,INTERACTED\_WITH)

#### NODES: Person

Cada persona en el sistema es representa com un node `:Person` amb els atributs següents:

Atribut	Tipus	Descripció
<code>name</code>	<code>string</code>	Nom de la persona
<code>age</code>	<code>integer</code>	Edat
<code>gender</code>	<code>string</code>	Gènere (ex. "M", "F", "NB"...)
<code>interests</code>	<code>list</code>	Llista d'interessos (ex. viatges, art...)

#### RELACIONS

##### 1. [ `:FRIEND` ]

Relació d'amistat o connexió social entre dues persones.

- **Motivació:** permet detectar connexions indirectes del tipus "amic d'un amic", útil per filtrar suggeriments que no siguin desconeguts totals.
- **Tipus:** dirigida o bidireccional (es pot duplicar si cal simular bidireccionalitat).

##### 2. [ `:DATED` ]

Indica que dues persones han tingut una relació romàntica en el passat.

- **Motivació:** Serveix per excloure "ex" de recomanacions, o per analitzar patrons de relacions prèvies.
- **Nota:** Es pot expandir amb dates, estat, o motius de trencament si calgués en el futur.

### 3. [:INTERACTED\_WITH {type, timestamp}]

Relació que representa una interacció concreta entre dues persones: "like", missatge, comentari...

- **Motivació:** Dona informació temporal i qualitativa sobre el nivell d'interès o contacte entre usuaris.
- **Atributs:**
  - **type:** string ("like", "message", etc.)
  - **timestamp:** data i hora de la interacció

## 2. Crear un script para poblar la base con algunos datos de prueba ( usando Cypher)

Creemos un archivo llamado **seed.py** (fuera del **venv**) con este contenido:

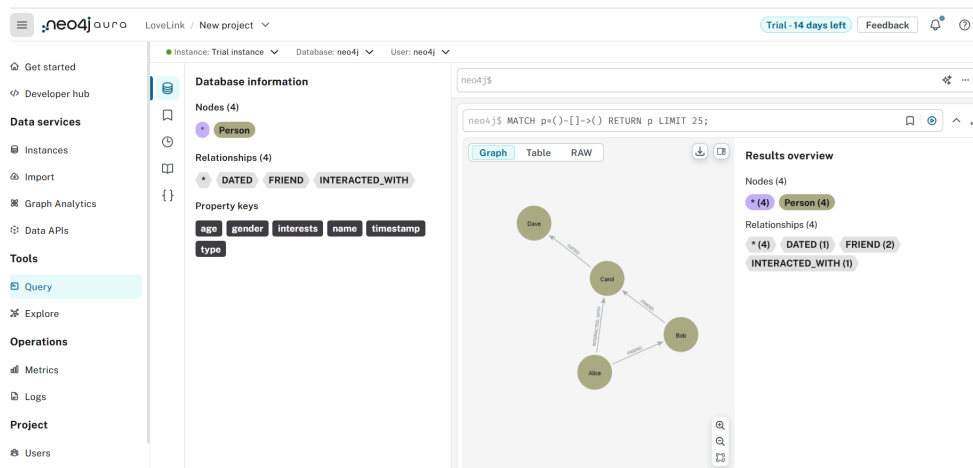
```
seed.py U X
seed.py > ...
1  from database import run_query
2
3  def seed_data():
4      query = """
5          MERGE (alice:Person {name: "Alice", age: 30, gender: "F", interests: ["cine", "viatges"]})
6          MERGE (bob:Person {name: "Bob", age: 32, gender: "M", interests: ["esport", "música"]})
7          MERGE (carol:Person {name: "Carol", age: 28, gender: "F", interests: ["lectura", "art"]})
8          MERGE (dave:Person {name: "Dave", age: 35, gender: "M", interests: ["viatges", "tecnologia"]})
9
10         MERGE (alice)-[:FRIEND]->(bob)
11         MERGE (bob)-[:FRIEND]->(carol)
12         MERGE (carol)-[:DATED]->(dave)
13         MERGE (alice)-[:INTERACTED_WITH {type: "like", timestamp: datetime()}]->(carol)
14         """
15         run_query(query)
16         print("Dades inicials creades correctament.")
17
18 if __name__ == "__main__":
19     seed_data()
20
```

y en la terminal:

python seed.py

```
PS C:\Users\Axel\Desktop\Computacio Distribuida i Aplicacions\PR2\LoveLink> python seed.py
Dades inicials creades correctament.
```

y se ha creado todo perfectamente.



### 3. Crear funciones para insertar personas y relaciones desde la API

Creamos el archivo [models.py](#) y añadimos esto:

```
models.py U X
models.py > ...
1  # models.py
2  from pydantic import BaseModel, Field
3  from typing import List, Literal, Optional
4  from datetime import datetime
5
6  class PersonCreate(BaseModel):
7      name: str
8      age: int
9      gender: str
10     interests: List[str]
11
12     class RelationshipType(str):
13         FRIEND = "FRIEND"
14         DATED = "DATED"
15         INTERACTED_WITH = "INTERACTED_WITH"
16
17     class RelationshipCreate(BaseModel):
18         from_person: str
19         to_person: str
20         type: Literal["FRIEND", "DATED", "INTERACTED_WITH"]
21         interaction_type: Optional[str] = None # solo si INTERACTED_WITH
22         timestamp: Optional[datetime] = None # solo si INTERACTED_WITH
23
```

Editamos el `main.py` y añadimos esto debajo de la sección donde tienes configurado el cliente de Neo4j:

```
# --- Endpoint para crear una persona ---
@app.post("/person/")
async def create_person(person: PersonCreate):
    query = """
    CREATE (:Person {name: $name, age: $age, gender: $gender, interests: $interests})
    """
    with driver.session() as session:
        session.run(query,
            name=person.name,
            age=person.age,
            gender=person.gender,
            interests=person.interests
        )
    return {"message": f"Persona {person.name} creada."}
```

```
# --- Endpoint para crear una relación ---
@app.post("/relationship/")
async def create_relationship(rel: RelationshipCreate):
    if rel.type == "INTERACTED_WITH" and (rel.interaction_type is None or rel.timestamp is None):
        raise HTTPException(status_code=400, detail="INTERACTED_WITH requiere interaction_type y timestamp.")

    match_query = """
    MATCH (a:Person {name: $from_name}), (b:Person {name: $to_name})
    """

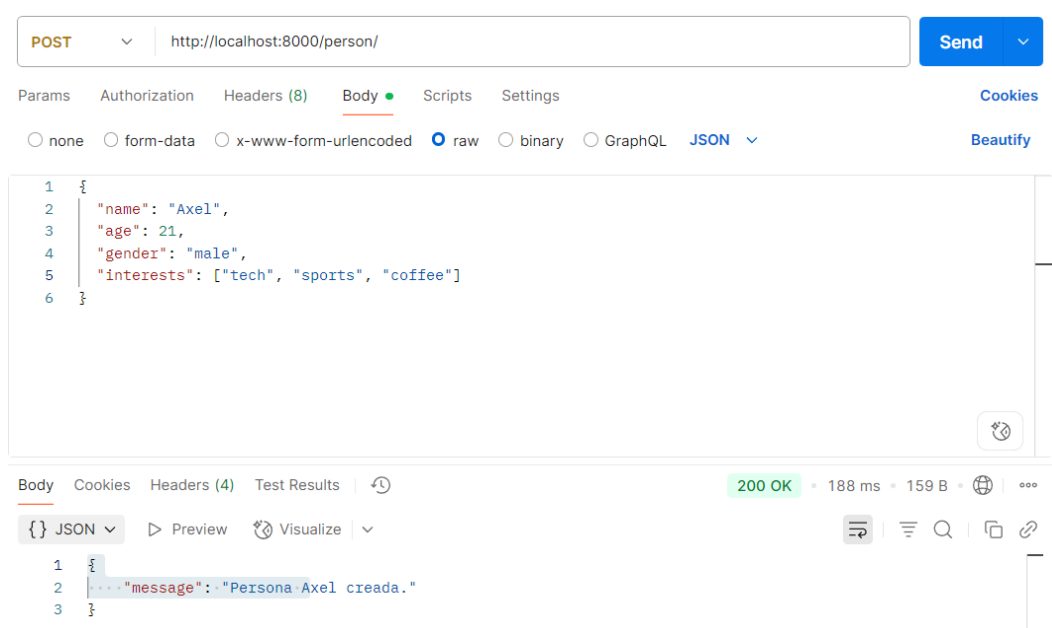
    if rel.type == "FRIEND":
        relation_query = "CREATE (a)-[:FRIEND]->(b)"
    elif rel.type == "DATED":
        relation_query = "CREATE (a)-[:DATED]->(b)"
    elif rel.type == "INTERACTED_WITH":
        relation_query = """
        CREATE (a)-[:INTERACTED_WITH {type: $interaction_type, timestamp: $timestamp}]->(b)
        """
    else:
        raise HTTPException(status_code=400, detail="Tipo de relación no soportado.")

    with driver.session() as session:
        session.run(match_query + relation_query,
            from_name=rel.from_person,
            to_name=rel.to_person,
            interaction_type=rel.interaction_type,
            timestamp=rel.timestamp
        )

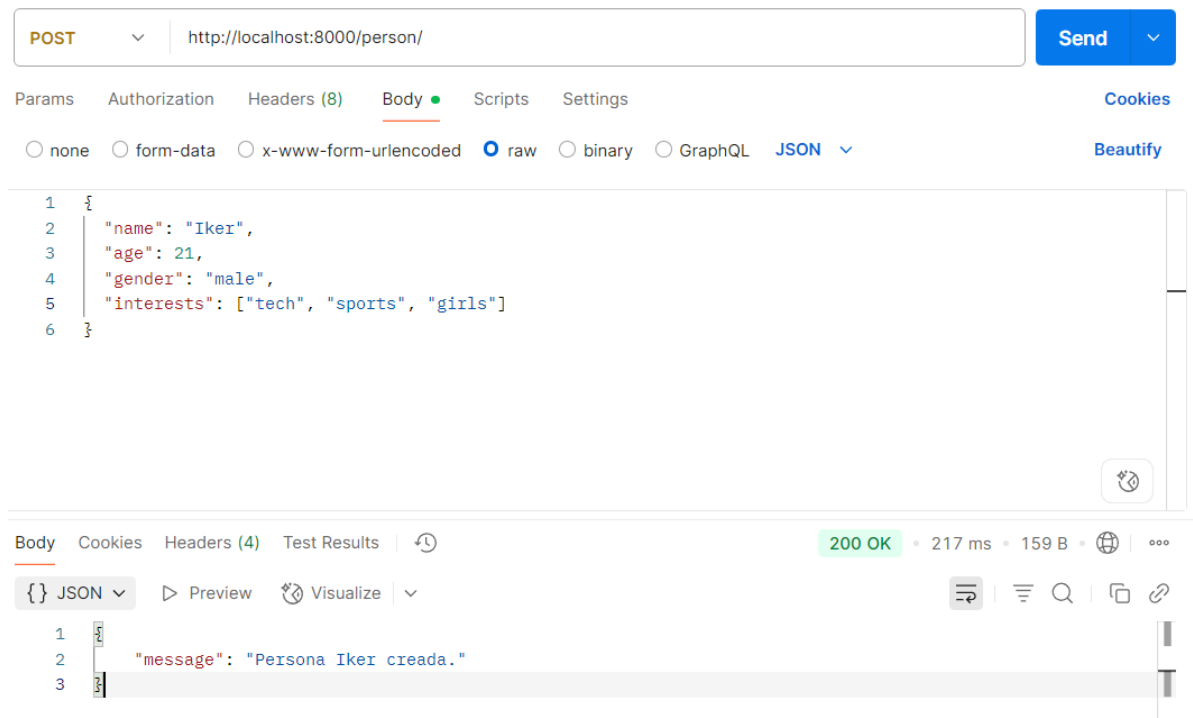
    return {"message": f"Relación {rel.type} creada entre {rel.from_person} y {rel.to_person}"}
```

Y hacemos pruebas con el postman:

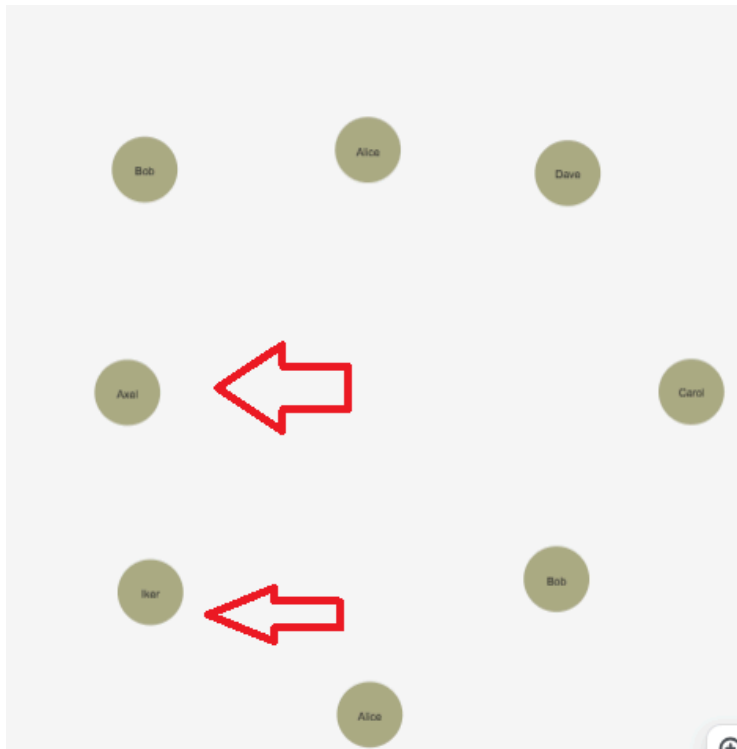
creamos al usuario 'Axel'



creamos al usuario 'Iker'



vemos que se crean perfectamente en la base de datos:



Y probamos a crear una relación de amistad entre ellos:

POST

Params Authorization Headers (8) **Body** Scripts Settings Cookies Beautify

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL JSON

```
1 {
2   "from_person": "Axel",
3   "to_person": "Iker",
4   "type": "FRIEND"
5 }
```

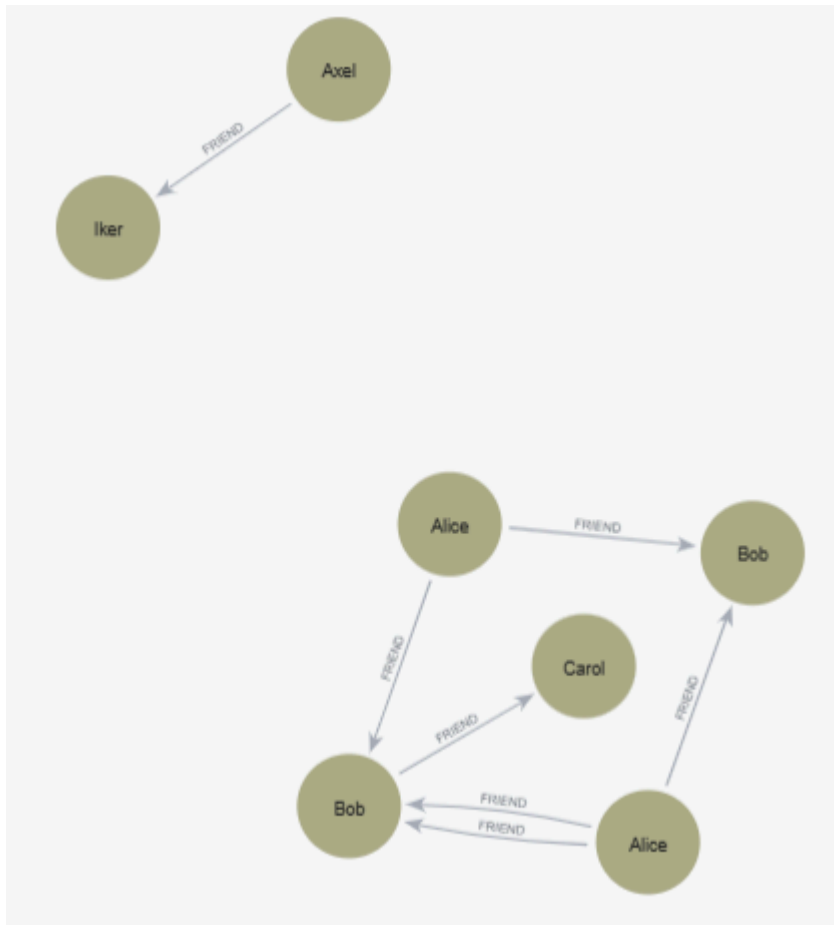
Body Cookies Headers (4) Test Results

200 OK • 213 ms • 180 B •

{ } JSON

```
1 {
2   "message": "Relación FRIEND creada entre Axel y Iker"
3 }
```

Y comprobamos que también se ha creado:



## FASE 4: Lógica de Recomendación

### 1. Diseñar y probar algoritmo “friend of a friend con twist romántico”

El objetivo es dar una lista de personas que **no estén conectadas directamente** (ni por **FRIEND**, **DATED**, **INTERACTED\_WITH**) pero que sean buenas candidatas por proximidad en el grafo.

#### Condiciones del algoritmo:

1. La persona a recomendar (**candidate**) no debe haber tenido ninguna relación previa con la persona a la que le salen las recomendaciones.
2. Debe estar a 1 o 2 saltos de distancia (pero no directamente conectado).
3. Opcional: podría ordenar por número de intereses en común (más adelante).

Lo siguiente es modificar nuestro [database.py](#) para añadir el algoritmo:

```
# Función de utilidad para buscar recomendaciones
def get_recommendations_for(name: str):
    query = """
    MATCH (me:Person {name: $name})
    MATCH (me)-[:FRIEND|DATED|INTERACTED_WITH*1..2]-(candidate:Person)
    WHERE me <> candidate
    AND NOT (me)-[:FRIEND|DATED|INTERACTED_WITH]-(candidate)
    RETURN DISTINCT candidate.name AS name,
           candidate.age AS age,
           candidate.gender AS gender,
           candidate.interests AS interests
    LIMIT 10
    """

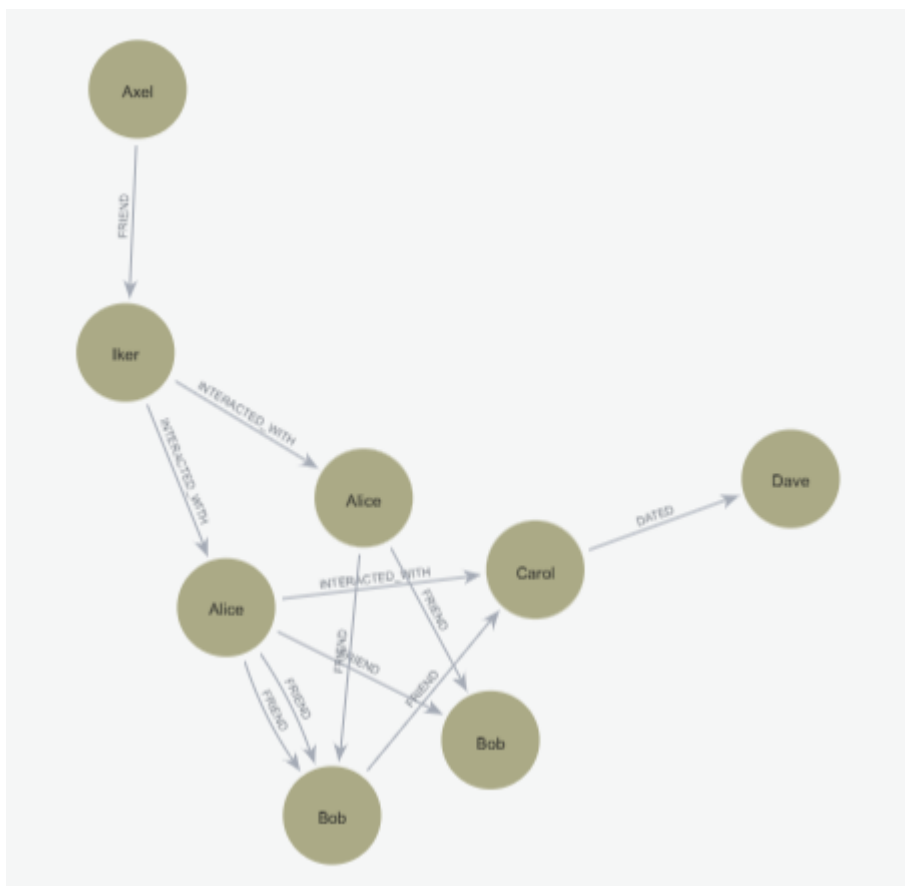
    with driver.session() as session:
        result = session.run(query, name=name)
        return [record.data() for record in result]
```



Ahora en el `main.py`, creamos el endpoint:

```
# --- Endpoint para obtener la recomendación ---
@app.get("/recommendations/{name}")
def get_recommendations(name: str):
    try:
        recommendations = get_recommendations_for(name)
        return {"recommendations": recommendations}
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))
```

Y vamos a probar:



según el grafo y el algoritmo, al usuario 'Axel', se le debería recomendar las usuarias "Alice"

probamos en el postman:

The screenshot shows the Postman interface with a GET request to `http://127.0.0.1:8000/recommendations/Axel`. The response is a 200 OK status with a response time of 294 ms and a body size of 299 B. The response body is a JSON array of two recommendation objects for the user "Alice".

```
{
  "recommendations": [
    {
      "name": "Alice",
      "age": 30,
      "gender": "F",
      "interests": [
        "cine",
        "viatges"
      ]
    },
    {
      "name": "Alice",
      "age": 28,
      "gender": "female",
      "interests": [
        "art",
        "hiking",
        "coffee"
      ]
    }
  ]
}
```

y efectivamente, se le recomiendan las usuarias “Alice”.

## 2. Añadir filtros opcionales: edad, género, intereses

La petición actual en `get_recommendations_for(name)` ya trae los candidatos. Ahora queremos:

- Comparar `user.interests` con `candidate.interests`
- Contar cuántos intereses tienen en común
- Ordenar por ese número (de mayor a menor)

Neo4j no tiene una función nativa para comparar listas directamente, pero se puede hacer con list operations.

Entonces modificaremos la función en `database.py` así:

```
# Función de utilidad para buscar recomendaciones
def get_recommendations_for(name: str):
    query = """
    MATCH (me:Person {name: $name})
    MATCH (me)-[:FRIEND|DATED|INTERACTED_WITH*1..2]-(candidate:Person)
    WHERE me <> candidate
    AND NOT (me)-[:FRIEND|DATED|INTERACTED_WITH]-(candidate)
    WITH me, candidate,
    [i IN me.interests WHERE i IN candidate.interests] AS common_interests
    RETURN candidate.name AS name,
           candidate.age AS age,
           candidate.gender AS gender,
           candidate.interests AS interests,
           size(common_interests) AS common_count
    ORDER BY common_count DESC
    LIMIT 10
    """

    with driver.session() as session:
        result = session.run(query, name=name)
        return [record.data() for record in result]
```

Esta función actualmente solo filtra por intereses.

Volvemos a hacer la petición de recommendations con el usuario 'Axel':

The screenshot shows a REST client interface with the following details:

- Method:** GET
- URL:** http://127.0.0.1:8000/recommendations/Axel
- Status:** 200 OK
- Response Time:** 526 ms
- Response Size:** 333 B

The JSON response body is as follows:

```
{
  "recommendations": [
    {
      "name": "Alice",
      "age": 28,
      "gender": "female",
      "interests": [
        "art",
        "hiking",
        "coffee"
      ],
      "common_count": 1
    },
    {
      "name": "Alice",
      "age": 30,
      "gender": "F",
      "interests": [
        "cine",
        "viatdes"
      ]
    }
  ]
}
```

Y podemos ver que ahora ha ordenado a la user 'Alice' de 28 años primero porque la edad es más cercana a la del user 'Axel' (21 años).

Con esto ya tenemos un buen algoritmo de recomendaciones, pero nosotros queremos más, entonces lo que hemos hecho es que el algoritmo te ayude a relacionarte (por gustos, intereses, edad, etc), pero que además tú puedas elegir a una persona con la que te interese relacionarte, y que el algoritmo te ayude con ello.

Para eso hemos hecho las siguientes implementaciones:

Añadimos esto en `models.py`:

```
class InterestCreate(BaseModel):
    from_person: str
    to_person: str
```

que es para crear la relación de interés.

Seguimos creando el endpoint para marcar interés (`main.py`):

```
# --- Endpoint para marcar interés ---
@app.post("/interest/")
async def express_interest(interest: InterestCreate):
    query = """
    MATCH (a:Person {name: $from_name}), (b:Person {name: $to_name})
    MERGE (a)-[r:INTERESTED_IN]->(b)
    RETURN a, b
    """

    with driver.session() as session:
        session.run(query, from_name=interest.from_person, to_name=interest.to_person)

    # Ahora chequeamos si hay interés mutuo para marcar un "match"
    match_check_query = """
    MATCH (a:Person {name: $from_name})-[:INTERESTED_IN]->(b:Person {name: $to_name}),
    (b)-[:INTERESTED_IN]->(a)
    RETURN a, b
    """

    with driver.session() as session:
        result = session.run(match_check_query, from_name=interest.from_person, to_name=interest.to_person)
        if result.single():
            return {"message": f"¡Es un match entre {interest.from_person} y {interest.to_person}!"}

    return {"message": f"{interest.from_person} ha mostrado interés en {interest.to_person}."}
```

Y también hemos creado el endpoint de mostrar matches del usuario ([main.py](#)):

```
# --- Endpoint para mostrar matches ---
@app.get("/matches/{name}")
async def get_matches(name: str):
    query = """
    MATCH (a:Person {name: $name})-[:INTERESTED_IN]->(b:Person),
    (b)-[:INTERESTED_IN]->(a)
    RETURN b.name AS name, b.age AS age, b.gender AS gender, b.interests AS interests
    """

    with driver.session() as session:
        result = session.run(query, name=name)
        return [record.data() for record in result]
```

y una vez añadida esta lógica, hemos actualizado el [database.py](#) para que el algoritmo de recomendaciones utilice esta lógica.

```
# Función de utilidad para buscar recomendaciones
def get_recommendations_for(name: str):
    query = """
    MATCH (me:Person {name: $name})

    // Buscar candidatos cercanos (máximo 2 saltos de relaciones sociales)
    MATCH path = (me)-[:FRIEND|DATED|INTERACTED_WITH*1..2]-(candidate:Person)

    WHERE me <> candidate
    AND NOT (me)-[:FRIEND|DATED|INTERACTED_WITH]-(candidate) // No recomendados si ya hay relación

    // Obtener intereses en común
    WITH me, candidate,
    [i IN me.interests WHERE i IN candidate.interests] AS common_interests,
    length(path) AS degree

    RETURN DISTINCT
    candidate.name AS name,
    candidate.age AS age,
    candidate.gender AS gender,
    candidate.interests AS interests,
    size(common_interests) AS common_count,
    degree,
    'Coincidís en ' + toString(size(common_interests)) + ' intereses' AS reason

    ORDER BY common_count DESC, degree ASC
    LIMIT 10
    """

    with driver.session() as session:
        result = session.run(query, name=name)
        return [record.data() for record in result]
```

Finalmente, hemos añadido la una función que te muestra el camino más corto hacía esa persona.

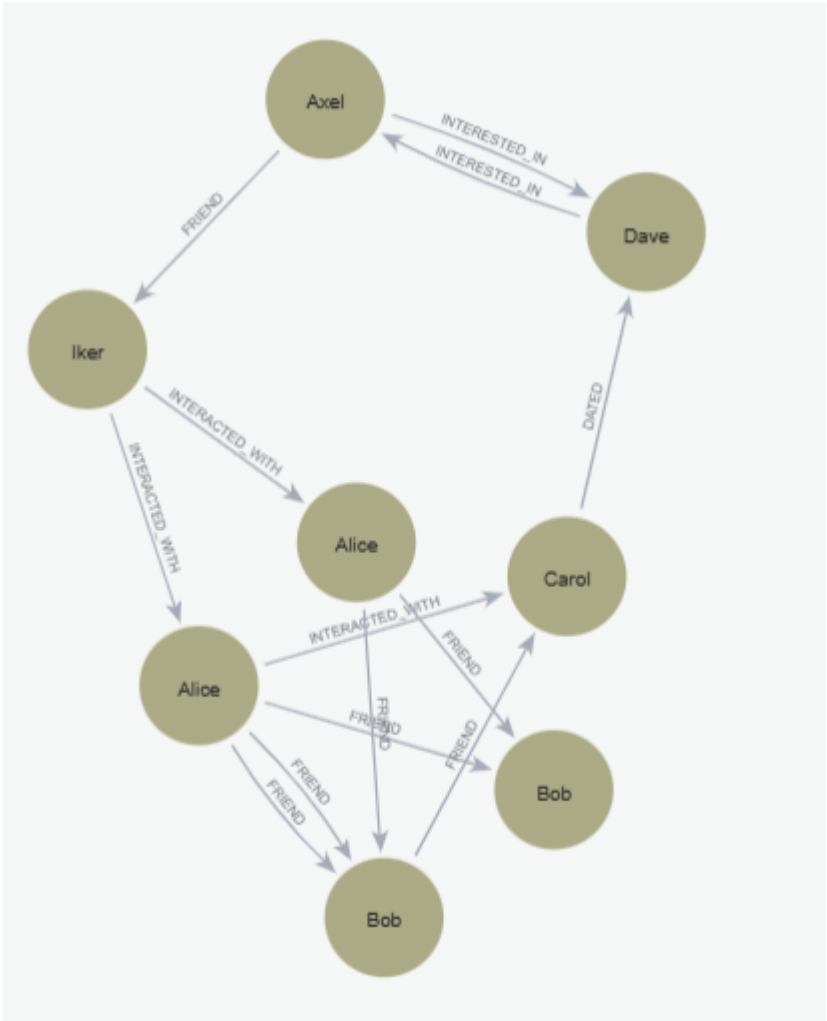
Añadimos la función en `database.py`:

```
# Función que te da el camino más corto hacía la persona que te interesa.
def path_to_person(from_name: str, to_name: str):
    query = """
    MATCH path = shortestPath(
      (from:Person {name: $from_name})-[:FRIEND|DATED|INTERACTED_WITH*..5]-(to:Person {name: $to_name})
    )
    RETURN [n IN nodes(path) | n.name] AS path_names,
           [rel IN relationships(path) | type(rel)] AS relationship_types
    """
    with driver.session() as session:
        result = session.run(query, from_name=from_name, to_name=to_name)
        record = result.single()
        if not record:
            return {"path": [], "types": []}
        return {
            "path": record["path_names"],
            "types": record["relationship_types"]
        }
```

y añadimos el endpoint en `main.py` para usar la función:

```
# --- Endpoint para mostrar el camino más corto hacia la persona que te interesa. ---
@app.get("/path-to/{from_name}/{to_name}")
def get_path_to_person(from_name: str, to_name: str):
    try:
        path = path_to_person(from_name, to_name)
        return path
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))
```

Y podemos ver te devuelve el camino de 'Axel' hasta 'Dave':



GET

http://127.0.0.1:8000/path-to/Axel/Dave

Send

Params

Authorization

Headers (6)

Body

Scripts

Settings

Cookies

Query Params

	Key	Value	Description	...	Bulk Edit
	Key	Value	Description		

Body

Cookies

Headers (4)

Test Results

200 OK

310 ms

236 B

...

JSON

Preview

Visualize

```
1  {
2    "path": [
3      "Axel",
4      "Iker",
5      "Alice",
6      "Carol",
7      "Dave"
8    ],
9    "types": [
10     "FRIEND",
11     "INTERACTED_WITH",
12     "INTERACTED_WITH",
13     "DATED"
14   ]
15 }
```

# FASE 5: Cliente web sencillo

## 1. Reestructura mínima del proyecto

añadimos las carpetas `static/` y `templates/`:

```
LoveLink/
├── static/
│   └── script.js    <- Lógica del cliente en JS
├── templates/
│   └── index.html   <- Página principal
├── main.py          <- Añadiremos renderizado de HTML aquí
└── ...
```

## 2. Añadir soporte para HTML en FastAPI

Modificamos `main.py` para servir la interfaz web:

- Importamos las nuevas librerías

```
from fastapi import FastAPI, HTTPException, Request
from fastapi.staticfiles import StaticFiles
from fastapi.templating import Jinja2Templates
from fastapi.responses import HTMLResponse
```

- Añadimos el soporte para archivos estáticos y plantillas

```
# Soporte para archivos estáticos y plantillas
app.mount("/static", StaticFiles(directory="static"), name="static")
templates = Jinja2Templates(directory="templates")

@app.get("/", response_class=HTMLResponse)
def index(request: Request):
    return templates.TemplateResponse("index.html", {"request": request})
```



### 3. Creamos el archivo **index.html** en la carpeta **templates**:

```
index.html U X
templates > index.html > ...
1 <!DOCTYPE html>
2 <html lang="es">
3 <head>
4   <meta charset="UTF-8" />
5   <title>LoveLink</title>
6   <link rel="stylesheet" href="/static/styles.css" />
7   <script defer src="/static/script.js"></script>
8 </head>
9 <body>
10  <h1>💖 LoveLink</h1>
11
12  <section>
13    <h2>🔍 Buscar recomendaciones</h2>
14    <input id="rec-name" placeholder="Tu nombre" />
15    <button onclick="getRecommendations()">Ver recomendaciones</button>
16    <div id="recommendation-list" class="card-container"></div>
17  </section>
18
19  <section>
20    <h2>🗺 Camino hacia alguien</h2>
21    <input id="from-name" placeholder="Desde" />
22    <input id="to-name" placeholder="Hasta" />
23    <button onclick="getPath()">Buscar camino</button>
24    <div id="path-results" class="path-container"></div>
25  </section>
26 </body>
27 </html>
28
```

### 3. Creamos el archivo **script.js** en la carpeta **static**:

```
JS script.js U X
static > JS script.js > ...
1 async function getRecommendations() {
2   const name = document.getElementById("rec-name").value;
3   const res = await fetch(`/recommendations/${name}`);
4   const data = await res.json();
5
6   const list = document.getElementById("recommendation-list");
7   list.innerHTML = "";
8
9   data.recommendations.forEach(person => {
10     const card = document.createElement("div");
11     card.className = "card";
12
13     if (typeof person === "string") {
14       card.textContent = person;
15     } else {
16       card.innerHTML = `
17         <strong>${person.name}</strong><br>
18         Edad: ${person.age}<br>
19         Género: ${person.gender}<br>
20         Intereses: ${person.interests.join(", ")}
21       `;
22     }
23
24     list.appendChild(card);
25   });
26 }
```

```

async function getPath() {
  const from = document.getElementById("from-name").value;
  const to = document.getElementById("to-name").value;
  const res = await fetch(`/path-to/${from}/${to}`);
  const data = await res.json();

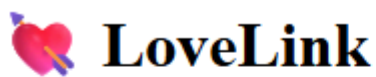
  const list = document.getElementById("path-results");
  list.innerHTML = "";

  data.path.forEach((name, i) => {
    const node = document.createElement("div");
    node.className = "path-node";
    node.textContent = name;
    list.appendChild(node);

    if (i < data.types.length) {
      const arrow = document.createElement("div");
      arrow.className = "path-arrow";
      arrow.innerHTML = `→ <em>${data.types[i]}</em>`;
      list.appendChild(arrow);
    }
  });
}

```

Y comprobamos que funciona nuestra humilde página web.



## Buscar recomendaciones

- Bob (30 años, male) - Intereses: tech, sports, coffee
- Bob (32 años, M) - Intereses: esport, música
- Carol (28 años, F) - Intereses: lectura, art


## Camino hacia alguien


- Iker
- Alice (INTERACTED\_WITH)
- Carol (INTERACTED\_WITH)

Añadimos un poco de css para dejarlo bonito:

```
# styles.css U X
static > # styles.css > ...
1  body {
2      font-family: 'Segoe UI', sans-serif;
3      background: #f9f9fc;
4      color: #333;
5      padding: 20px;
6      max-width: 800px;
7      margin: auto;
8  }
9
10 h1 {
11     text-align: center;
12     color: #d63384;
13 }
14
15 section {
16     margin-top: 30px;
17 }
18
19 input {
20     padding: 8px;
21     margin: 5px;
22     border: 1px solid #ccc;
23     border-radius: 6px;
24 }
25
26 button {
27     padding: 8px 14px;
28     background-color: #d63384;
29     color: white;
30     border: none;
31     border-radius: 6px;
32     cursor: pointer;
33 }
```

y vemos que ahora queda un poco más chula.



 **Buscar recomendaciones**

Ver recomendaciones

**Bob**  
Edad: 30  
Género: male  
Intereses: tech, sports, coffee

**Bob**  
Edad: 32  
Género: M  
Intereses: esport, música

**Carol**  
Edad: 28  
Género: F  
Intereses: lectura, art

 **Camino hacia alguien**

Buscar camino

Iker

 → INTERACTED\_WITH 

Alice

 → INTERACTED\_WITH 

Carol

# FASE 6: Autenticación y usuarios

## 1. Registro

Agrega esto a final de `requirements.txt`:

```
sqlalchemy>=1.4,<2.0
bcrypt==4.1.3
python-jose[cryptography]==3.3.0
passlib==1.7.4
```

Y ejecuta:

```
pip install -r requirements.txt
```

Creamos la carpeta `auth` con el archivo `users_db.py` añadimos el código:

```
users_db.py X
auth > users_db.py > create_user
1 # auth/users_db.py
2 from passlib.context import CryptContext
3 from neo4j import Driver
4
5 pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")
6
7 def create_user(driver: Driver, email: str, password: str, name: str, age: int, gender: str, interests: list):
8     hashed_password = pwd_context.hash(password)
9
10    with driver.session() as session:
11        # Verificar si ya existe
12        check = session.run("MATCH (p:Person {email: $email}) RETURN p", email=email).single()
13        if check:
14            raise ValueError("El usuario ya existe")
15
16        # Crear el nodo persona con todos los campos
17        session.run("""
18            CREATE (p:Person {
19                email: $email,
20                password: $password,
21                name: $name,
22                age: $age,
23                gender: $gender,
24                interests: $interests
25            })
26        """, {
27            "email": email,
28            "password": hashed_password,
29            "name": name,
30            "age": age,
31            "gender": gender,
32            "interests": interests
33        })
34
```

Creamos un nuevo modelo en `models.py`:

```
class UserCreate(BaseModel):
    email: EmailStr
    password: str
    name: str
    age: int
    gender: str
    interests: List[str]
```

Y finalmente abrimos `main.py` para importar el modelo y la función:

```
from models import RelationshipCreate, InterestCreate, UserCreate
from auth.users_db import create_user
```

Y añadimos el endpoint para registrarlos:

```
# --- Endpoint para registrarlos ---
@app.post("/register")
def register(user: UserCreate):
    try:
        create_user(driver, user.email, user.password, user.name, user.age, user.gender, user.interests)
        return {"message": "Usuario creado correctamente"}
    except ValueError as e:
        raise HTTPException(status_code=400, detail=str(e))
    except Exception as e:
        raise HTTPException(status_code=500, detail="Error interno del servidor")
```

Para evitar líos, hemos borrado el endpoint de **CrearPerson**. Pero lo puedes dejar sin problema. También hemos borrado el modelo **Person** de `models.py`. Tampoco lo vamos a usar más.

## 2. Login

Creamos el archivo `jwt_handler.py` en la carpeta `auth`. Este archivo se encargará de crear y verificar los tokens JWT.

```
jwt_handler.py U X
auth > jwt_handler.py > ...
1 # auth/jwt_handler.py
2 from datetime import datetime, timedelta
3 from jose import JWTError, jwt
4 from fastapi import HTTPException, status, Depends
5 from fastapi.security import OAuth2PasswordBearer
6 from typing import Optional
7 import os
8 from dotenv import load_dotenv
9
10 load_dotenv()
11
12 SECRET_KEY = os.getenv("SECRET_KEY", "secret") # Usa una clave real en producción
13 ALGORITHM = "HS256"
14 ACCESS_TOKEN_EXPIRE_MINUTES = 60
15
16 oauth2_scheme = OAuth2PasswordBearer(tokenUrl="login")
17
18 def create_access_token(data: dict, expires_delta: Optional[timedelta] = None):
19     to_encode = data.copy()
20     expire = datetime.utcnow() + (expires_delta or timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES))
21     to_encode.update({"exp": expire})
22     encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
23     return encoded_jwt
24
25 def decode_access_token(token: str = Depends(oauth2_scheme)):
26     try:
27         payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
28         return payload
29     except JWTError:
30         raise HTTPException(
31             status_code=status.HTTP_401_UNAUTHORIZED,
32             detail="Token inválido o expirado",
33             headers={"WWW-Authenticate": "Bearer"},
34         )
```

Luego creamos el archivo `login.py` también en la carpeta `auth`. Este archivo maneja el proceso de login.

```
login.py M X
auth > login.py > ...
1 # auth/login.py Este archivo maneja el proceso de login.
2
3 from fastapi import Depends, HTTPException, status
4 from fastapi.security import OAuth2PasswordRequestForm
5 from auth.users_db import get_user_by_email, verify_password
6 from auth.jwt_handler import create_access_token
7 from neo4j import Driver
8 from fastapi.responses import JSONResponse
9
10 def login_user(form_data: OAuth2PasswordRequestForm, driver: Driver):
11     user = get_user_by_email(driver, form_data.username)
12     if not user or not verify_password(form_data.password, user["password"]):
13         raise HTTPException(
14             status_code=status.HTTP_401_UNAUTHORIZED,
15             detail="Credenciales inválidas"
16         )
17
18     token = create_access_token({"sub": user["email"], "name": user["name"]})
19
20     # Guardar el token como cookie segura
21     response = JSONResponse(content={"message": "Login correcto"})
22     response.set_cookie(
23         key="access_token",
24         value=token,
25         httponly=True, # Evita acceso vía JS
26         secure=False,
27         samesite="lax"
28     )
29     return response
```

Creamos las funciones `get_user_by_email` y `verify_password` en `users_db.py`:

```
68 # 🔒 Obtener usuario por email
69 def get_user_by_email(driver: Driver, email: str):
70     with driver.session() as session:
71         result = session.run("MATCH (p:Person {email: $email}) RETURN p", email=email)
72         record = result.single()
73         if record:
74             return record["p"]
75         return None
76
77 # 🔒 Verificar contraseña (plain vs hashed)
78 def verify_password(plain_password: str, hashed_password: str) -> bool:
79     return pwd_context.verify(plain_password, hashed_password)
```

Y como siempre, importamos y usamos las funciones en `main.py`.

```
from auth.login import login_user
from fastapi.security import OAuth2PasswordRequestForm
from auth.jwt_handler import decode_access_token
```

```
# --- Endpoint para iniciar sesión ---
@app.post("/login")
def login(form_data: OAuth2PasswordRequestForm = Depends()):
    return login_user(form_data, driver)

# --- Endpoint para obtener la información del usuario que ha iniciado sesión ---
@app.get("/me")
def get_current_user(token_data: dict = Depends(decode_access_token)):
    return {"email": token_data["sub"], "name": token_data.get("name")}
```



### 3. Views de Register y Login

añadimos función logout() en [script.js](#)

```
function logout() {  
  fetch("/logout", {  
    method: "GET",  
    credentials: "include"  
  }).then(() => {  
    window.location.href = "/login-page";  
  });  
}
```

añadimos endpoint para cerrar sesión, para que se te redirija a /login-page y borre eficazmente las cookies del navegador, ya que JS no puede hacerlo.

```
# --- Endpoint para cerrar sesión ---  
@app.get("/logout")  
def logout():  
    response = RedirectResponse(url="/login-page", status_code=303)  
    response.delete_cookie("access_token")  
    return response
```

Agregamos el middleware toda la configuración del frontend en main.py:

```
#FRONTEND  
  
# Middleware para proteger el acceso a /  
class AuthMiddleware(BaseHTTPMiddleware):  
    async def dispatch(self, request: Request, call_next):  
        if request.url.path == "/":  
            token = request.cookies.get("access_token")  
            if not token:  
                return RedirectResponse("/login-page")  
            try:  
                decode_access_token(token)  
            except Exception:  
                return RedirectResponse("/login-page")  
        return await call_next(request)  
  
app.add_middleware(AuthMiddleware)  
  
app.mount("/static", StaticFiles(directory="static"), name="static")  
templates = Jinja2Templates(directory="templates")  
  
@app.get("/", response_class=HTMLResponse)  
def index(request: Request):  
    return templates.TemplateResponse("index.html", {"request": request})  
  
@app.get("/login-page", response_class=HTMLResponse)  
def login_page(request: Request):  
    return templates.TemplateResponse("login.html", {"request": request})  
  
@app.get("/register-page", response_class=HTMLResponse)  
def register_page(request: Request):  
    return templates.TemplateResponse("register.html", {"request": request})
```

Creamos el login.html:

```
login.html U X
templates > login.html > ...
1  <!DOCTYPE html>
2  <html>
3  <head>
4  | <title>Login - LoveLink</title>
5  </head>
6  <body>
7  | <h1>Iniciar Sesión</h1>
8
9  | <form id="login-form">
10 |   <input type="email" id="email" placeholder="Correo" required><br>
11 |   <input type="password" id="password" placeholder="Contraseña" required><br>
12 |   <button type="submit">Iniciar sesión</button>
13 | </form>
14
15 | <!-- Botón para ir al registro -->
16 | <p><?No tienes cuenta? <a href="/register-page">Regístrate aquí</a></p>
17
18 | <script>
19 |   document.getElementById("login-form").addEventListener("submit", async (e) => {
20 |     e.preventDefault();
21
22 |     const formData = new URLSearchParams();
23 |     formData.append("username", document.getElementById("email").value);
24 |     formData.append("password", document.getElementById("password").value);
25
26 |     const res = await fetch("/login", {
27 |       method: "POST",
28 |       headers: {
29 |         "Content-Type": "application/x-www-form-urlencoded"
30 |       },
31 |       body: formData,
32 |       credentials: "include"
33 |     });
34
35 |     if (res.ok) {
36 |       window.location.href = "/";
37 |     } else {
38 |       alert("Credenciales incorrectas");
39 |     }
40 |   });
41 | </script>
42 </body>
43 </html>
```

el register.html:

```
register.html U X
templates > register.html > ...
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title>Registro - LoveLink</title>
5  </head>
6  <body>
7    <h1>Registrarse</h1>
8
9    <form id="register-form">
10     <input type="email" id="email" placeholder="Correo" required><br>
11     <input type="password" id="password" placeholder="Contraseña" required><br>
12     <input type="text" id="name" placeholder="Nombre" required><br>
13     <input type="number" id="age" placeholder="Edad" required><br>
14     <input type="text" id="gender" placeholder="Género" required><br>
15     <input type="text" id="interests" placeholder="Intereses (coma separada)" required><br>
16     <button type="submit">Registrarse</button>
17   </form>
18
19   <!-- Enlace para ir al login -->
20   <p>¿Ya tienes cuenta? <a href="/login-page">Inicia sesión aquí</a></p>
21
22   <script>
23     document.getElementById("register-form").addEventListener("submit", async (e) => {
24       e.preventDefault();
25
26       const body = {
27         email: document.getElementById("email").value,
28         password: document.getElementById("password").value,
29         name: document.getElementById("name").value,
30         age: parseInt(document.getElementById("age").value),
31         gender: document.getElementById("gender").value,
32         interests: document.getElementById("interests").value.split(",").map(i => i.trim())
33       };
34
35       const res = await fetch("/register", {
36         method: "POST",
37         headers: { "Content-Type": "application/json" },
38         body: JSON.stringify(body)
39       });
40
41       if (res.ok) {
42         alert("Usuario registrado. Inicia sesión.");
43         window.location.href = "/login-page";
44       } else {
45         alert("Error al registrarse.");
46       }
47     });
48   </script>
49 </body>
50 </html>
```

Actualizamos el index.html para que tenga el botón de logout:

```
index.html M X
templates > index.html > html > body > section
1 <!DOCTYPE html>
2 <html lang="es">
3 <head>
4   <meta charset="UTF-8" />
5   <title>LoveLink</title>
6   <link rel="stylesheet" href="/static/styles.css" />
7   <script defer src="/static/script.js"></script>
8 </head>
9 <body>
10  <h1>❤️ LoveLink</h1>
11  <button onclick="logout()" style="float: right; margin-top: -50px;">Cerrar sesión</button>
12
13  <section>
14    <h2>🔍 Buscar recomendaciones</h2>
15    <input id="rec-name" placeholder="Tu nombre" />
16    <button onclick="getRecommendations()">Ver recomendaciones</button>
17    <div id="recommendation-list" class="card-container"></div>
18  </section>
19
20  <section>
21    <h2>🗺️ Camino hacia alguien</h2>
22    <input id="from-name" placeholder="Desde" />
23    <input id="to-name" placeholder="Hasta" />
24    <button onclick="getPath()">Buscar camino</button>
25    <div id="path-results" class="path-container"></div>
26  </section>
27 </body>
28 </html>
```

y finalmente, añadimos el css.

```
# auth.css U X
static > # auth.css > ...
1 /* LOGIN Y REGISTER CSS*/
2 *,
3 *:before,
4 *:after {
5   padding: 0;
6   margin: 0;
7   box-sizing: border-box;
8 }
9
10 body {
11   background-color: #080710;
12   font-family: 'Poppins', sans-serif;
13   height: 100vh;
14   overflow: hidden;
15 }
16
17 /* Fondo decorativo */
18 .background {
19   width: 430px;
```

## Login Page

← → ↻ 127.0.0.1:8000/login-page

### Iniciar Sesión

Correo

Contraseña

Iniciar sesión

¿No tienes cuenta? Regístrate aquí

## Register Page

← → ↻ 127.0.0.1:8000/register-page

### Registrarse

Correo

Contraseña

Nombre

Edad

Género

Intereses

Intereses (coma separada)

Registrarse

¿Ya tienes cuenta? Inicia sesión aquí

## Home Page

LoveLink

Cerrar sesión

### Buscar recomendaciones

Tu nombre

Ver recomendaciones

### Camino hacia alguien

Desde

Hasta

Buscar camino

# FASE 7: RED SOCIAL FUNCIONAL

## 1. Implementar las funciones para los usuarios logueados

A partir de aquí son fotos de las mejoras, implementaciones que hemos hecho en el código para dejar la página web más elegante, útil y bonita para los usuarios.

## RECOMMENDATIONS

script.js

```
// Función para cargar recomendaciones del user logeado
document.addEventListener("DOMContentLoaded", () => {
  loadMyRecommendations();
});

async function loadMyRecommendations() {
  try {
    const response = await fetch("/user-logged-recommendations", {
      method: "GET",
      credentials: "include"
    });

    if (!response.ok) {
      console.error("Error cargando recomendaciones", await response.text());
      return;
    }

    const data = await response.json();
    const list = document.getElementById("my-recommendation-list");
    list.innerHTML = "";

    if (!data.recommendations || data.recommendations.length === 0) {
      list.innerHTML = "<p style='color: white;'>No tienes recomendaciones aún.<br>Agrega a amigos para obtener recomendaciones.</p>";
      return;
    }

    data.recommendations.forEach(person => {
      const card = document.createElement("div");
      card.className = "card";

      if (typeof person === "string") {
        card.textContent = person;
      } else {
        card.innerHTML = `
        <strong>${person.name}</strong><br>
        Edad: ${person.age}<br>
        Género: ${person.gender}<br>
        Intereses: ${person.interests.join(", ")}<br>
        Motivo: ${person.reason}
        `;
      }

      list.appendChild(card);
    });
  } catch (error) {
    console.error("Error en la petición a /user-logged-recommendations:", error);
  }
}
```

script.js

```
// Función para leer cookies
function getCookie(name) {
  const match = document.cookie.match(new RegExp('(^| )' + name + '=[^;]+'));
  return match ? match[2] : null;
}
```

main.py


```
# --- Endpoint para coger las recomendaciones del usuario logeado ---
@app.get("/user-logged-recommendations")
def get_my_recommendations(user_node: dict = Depends(get_current_user)):
    name = user_node.get("name", "").strip()
    recommendations = get_recommendations_for(name)
    return {"recommendations": recommendations}
```

main.py

```
# --- Endpoint para obtener la información del usuario que ha iniciado sesión ---
def get_current_user(request: Request):
    token = request.cookies.get("access_token")
    if not token:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="No se encontró token en las cookies"
        )
    return decode_access_token(token)

@app.get("/me")
def get_current_user_info(user_data: dict = Depends(get_current_user)):
    return {"email": user_data["sub"], "name": user_data.get("name")}
```

funciona:



## ♥ Mis recomendaciones

<p><b>prueba3</b></p> <p>Edad: 69</p> <p>Género: M</p> <p>Intereses: prueba3, prueba3</p> <p>Motivo: Coincidís en 1 intereses</p>	<p><b>Bob</b></p> <p>Edad: 32</p> <p>Género: M</p> <p>Intereses: esport, música</p> <p>Motivo: Coincidís en 0 intereses</p>
<p><b>Bob</b></p> <p>Edad: 30</p> <p>Género: male</p> <p>Intereses: tech, sports, coffee</p> <p>Motivo: Coincidís en 0 intereses</p>	<p><b>me</b></p> <p>Edad: 90</p> <p>Género: Male</p> <p>Intereses: me</p> <p>Motivo: Coincidís en 0 intereses</p>
<p><b>Carol</b></p> <p>Edad: 28</p> <p>Género: F</p> <p>Intereses: lectura, art</p> <p>Motivo: Coincidís en 0 intereses</p>	<p><b>Iker</b></p> <p>Edad: 21</p> <p>Género: male</p> <p>Intereses: tech, sports, girls</p> <p>Motivo: Coincidís en 0 intereses</p>

## PATH-TO-PERSON

script.js

```
// Función para que el user loggeado pueda buscar el camino.
function renderPath(data) {
  const list = document.getElementById("path-results-logged");
  list.innerHTML = "";

  // Comprobar que data.path existe y tiene contenido
  if (!data.path || data.path.length === 0) {
    list.textContent = "Usuario no encontrado.";
    return;
  }

  data.path.forEach((name, i) => {
    const node = document.createElement("div");
    node.className = "path-node";
    node.textContent = name;
    list.appendChild(node);

    if (i < data.types.length) {
      const arrow = document.createElement("div");
      arrow.className = "path-arrow";
      arrow.innerHTML = `→ <em>${data.types[i]}</em>`;
      list.appendChild(arrow);
    }
  });
}

async function getPathFromLoggedUser() {
  const to = document.getElementById('to-name-logged').value.trim();
  if (!to) {
    alert('Por favor, introduce el nombre de la persona.');
```

return;

```
  }

  // Opcional: muestra mensaje mientras carga
  document.getElementById('path-results-logged').innerHTML = 'Cargando camino...';

  try {
    const res = await fetch(`/path-to-user/${encodeURIComponent(to)}`, {
      credentials: 'include'
    });

    if (!res.ok) {
      throw new Error(`Error en la petición: ${res.status}`);
    }

    const data = await res.json();


    // Aquí llama a la función que muestra el camino en pantalla
    renderPath(data, 'path-results-logged');
  } catch (error) {
    document.getElementById('path-results-logged').innerHTML = `Error: ${error.message}`;
  }
}
```



main.py

```
# --- Endpoint nuevo con from_name desde usuario logeado y to_name como parámetro ---
@app.get("/path-to-user/{to_name}")
def get_path_from_logged_user(to_name: str, user_data: dict = Depends(get_current_user)):
    from_name = user_data.get("name")
    if not from_name:
        raise HTTPException(status_code=400, detail="Usuario sin nombre válido en token")
    try:
        path = path_to_person(from_name, to_name)
        return path
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))
```

funciona:

 Encuentra tu camino hacia esa persona especial

prueba7

 → *FRIEND*

Alice

 → *INTERACTED\_WITH*

Carol

## PERFIL Y EDITAR TU PERFIL

script.js

```
// Función para ir a tu perfil.  
function goToProfile() {  
    window.location.href = "/profile";  
}
```

main.py

```
# --- Endpoint para ir al profile ---  
@app.get("/profile", response_class=HTMLResponse)  
def profile(request: Request):  
    return templates.TemplateResponse("profile.html", {"request": request})  
  
from fastapi.responses import JSONResponse  
from auth.jwt_handler import create_access_token, decode_access_token  
from fastapi import Request
```

index.html

```
<button class="profile-btn" onclick="goToProfile()">👤 Mi perfil</button>
```

creamos profile.html en la carpeta templates

```
profile.html U X  
templates > profile.html > html > head > style > input > select  
  
1 <!DOCTYPE html>  
2 <html lang="es">  
3 <head>  
4     <meta charset="UTF-8" />  
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />  
6     <title>Mi perfil - LoveLink</title>  
7     <link rel="stylesheet" href="/static/styles.css" />  
8     <style>  
9         * {  
10             box-sizing: border-box;  
11         }  
12  
13         body {  
14             background-color: #080710;  
15             font-family: 'Poppins', sans-serif;  
16             color: white;  
17             padding: 0;  
18             margin: 0;  
19             min-height: 100vh;  
20         }  
21  
22         .container {  
23             max-width: 500px;  
24             margin: 60px auto;  
25             padding: 30px;  
26             background-color: rgba(255, 255, 255, 0.1);  
27             backdrop-filter: blur(8px);  
28             border-radius: 12px;
```

main.py

```
# --- Endpoint para editar los datos del user ---
class UserUpdate(BaseModel):
    name: str
    age: int
    gender: str
    interests: List[str]

@app.put("/update-profile")
def update_profile(
    request: Request,
    data: UserUpdate,
    user_data: dict = Depends(get_current_user) # Usuario autenticado desde la cookie
):
    email = user_data.get("sub")
    if not email:
        raise HTTPException(status_code=401, detail="Token inválido")

    # Actualizar en la base de datos
    with driver.session() as session:
        result = session.run("""
            MATCH (p:Person {email: $email})
            SET p.name = $name,
                p.age = $age,
                p.gender = $gender,
                p.interests = $interests
            RETURN p
        """, {
            "email": email,
            "name": data.name,
            "age": data.age,
            "gender": data.gender,
            "interests": data.interests
        })
        updated = result.single()
        if not updated:
            raise HTTPException(status_code=404, detail="Usuario no encontrado")

    # Crear nuevo token con el nombre actualizado
    new_token = create_access_token({"sub": email, "name": data.name})

    # Devolver el nuevo token en la cookie
    response = JSONResponse({"message": "Perfil actualizado correctamente"})
    response.set_cookie(
        key="access_token",
        value=new_token,
        httponly=True,
        samesite="lax",
        secure=False # Cambia a True si usas HTTPS
    )

    return response
```

actualizamos y añadimos esto en profile.html

```
async function updateUserData(event) {
  event.preventDefault();
  const selectedInterests = Array.from(document.querySelectorAll(".interest-btn.selected"))
    .map(btn => btn.textContent);

  const payload = {
    name: document.getElementById("user-name").value,
    age: parseInt(document.getElementById("user-age").value),
    gender: document.getElementById("user-gender").value,
    interests: selectedInterests
  };

  const res = await fetch("/update-profile", {
    method: "PUT",
    headers: { "Content-Type": "application/json" },
    credentials: "include",
    body: JSON.stringify(payload)
  });

  if (res.ok) {
    alert("¡Perfil actualizado correctamente!");
  } else {
    alert("Error al actualizar perfil.");
  }
}

function goHome() {
  window.location.href = "/";
}

document.getElementById("edit-form").addEventListener("submit", updateUserData);
loadUserData();
```

funciona:

← → ↻ 127.0.0.1:8000/profile

### Mi Perfil

**Nombre:**

**Email:**

**Edad:**

**Género:**

**Intereses:**

Música

Cine

Viajes

Deporte

Lectura

Cocina

Tecnología

Naturaleza

Videojuegos

Arte


Animales

Fitness

Moda

Fotografía

Idiomas

 Guardar cambios

← Volver al inicio

## AÑADIR FOTOS

models.py:

```
class UserCreate(BaseModel):
    email: EmailStr
    password: str
    name: str
    age: int
    gender: str
    interests: List[str]
    profile_picture: Optional[str] = None # <---
```

actualizamos el create\_user de usersdb.py

```
def create_user(driver: Driver, email: str, password: str, name: str, age: int, gender: str, interests: list, profile_picture: str = ""):
    hashed_password = pwd_context.hash(password)

    with driver.session() as session:
        # Verificar si ya existe
        check = session.run("MATCH (p:Person {email: $email}) RETURN p", email=email).single()
        if check:
            raise ValueError("El usuario ya existe")

        # Crear el nodo persona
        session.run("""
        CREATE (p:Person {
            email: $email,
            password: $password,
            name: $name,
            age: $age,
            gender: $gender,
            interests: $interests,
            profile_picture: $profile_picture
        })
        """, {
            "email": email,
            "password": hashed_password,
            "name": name,
            "age": age,
            "gender": gender,
            "interests": interests,
            "profile_picture": profile_picture
        })
```

y añadimos

```
# 🗑 Obtener usuario por nombre
def get_user_by_name(driver: Driver, name: str):
    with driver.session() as session:
        result = session.run("MATCH (p:Person {name: $name}) RETURN p", name=name)
        record = result.single()
        if record:
            return record["p"]
        return None
```

## main.py

```
# --- Endpoint para editar los datos del user ---
class UserUpdate(BaseModel):
    name: str
    age: int
    gender: str
    interests: List[str]
    profile_picture: Optional[str] = None

@app.put("/update-profile")
async def update_profile(
    request: Request,
    name: str = Form(...),
    age: int = Form(...),
    gender: str = Form(...),
    interests: str = Form(...), # Viene como stringified JSON
    profile_picture: Optional[UploadFile] = File(None),
    user_data: dict = Depends(get_current_user)
):
    email = user_data.get("sub")
    if not email:
        raise HTTPException(status_code=401, detail="Token inválido")

    interests_list = json.loads(interests)

    # Guardar la imagen si existe
    picture_url = ""
    if profile_picture:
        ext = os.path.splitext(profile_picture.filename)[1]
        filename = f"{uuid.uuid4().hex}{ext}"
        filepath = os.path.join("static/uploads", filename)

        with open(filepath, "wb") as buffer:
            shutil.copyfileobj(profile_picture.file, buffer)

        picture_url = f"/static/uploads/{filename}"

    with driver.session() as session:
        result = session.run("""
            MATCH (p:Person {email: $email})
            SET p.name = $name,
                p.age = $age,
                p.gender = $gender,
                p.interests = $interests,
                p.profile_picture = CASE WHEN $picture <> '' THEN $picture ELSE p.profile_picture END
            RETURN p
        """, {
            "email": email,
            "name": name,
            "age": age,
            "gender": gender,
            "interests": interests_list,
            "picture": picture_url
        })
        updated = result.single()
        if not updated:
            raise HTTPException(status_code=404, detail="Usuario no encontrado")

    # Crear nuevo token
    new_token = create_access_token({"sub": email, "name": name})

    response = JSONResponse({"message": "Perfil actualizado correctamente"})
    response.set_cookie(key="access_token", value=new_token, httponly=True, samesite="lax", secure=False)
    return response
```

main.py

```
# --- Endpoint para coger las recomendaciones del usuario logeado ---
@app.get("/user-logged-recommendations")
def get_my_recommendations(user_node: dict = Depends(get_current_user)):
    name = user_node.get("name", "").strip()
    recommendations = get_recommendations_for(name)

    enriched_recommendations = []
    for person in recommendations:
        full_user = get_user_by_name(driver, person["name"])
        profile_picture = full_user.get("profile_picture", "") if full_user else ""

        person["profile_picture"] = profile_picture if profile_picture else "/static/default.jpg"
        enriched_recommendations.append(person)

    return {"recommendations": enriched_recommendations}
```

main.py

```
# --- Endpoint para saber quien soy (Cogiendo todos los datos reales de la database) ---
def get_user_by_email(driver: Driver, email: str):
    with driver.session() as session:
        result = session.run("MATCH (p:Person {email: $email}) RETURN p", email=email)
        record = result.single()
        if record:
            return record["p"]
        return None

@app.get("/whoami")
def whoami(request: Request):
    token = request.cookies.get("access_token")
    if not token:
        raise HTTPException(status_code=401, detail="No token provided")

    try:
        payload = decode_access_token(token)
        email = payload.get("sub")
        user_node = get_user_by_email(driver, email)
        if not user_node:
            raise HTTPException(status_code=404, detail="User not found")

        return {
            "name": user_node.get("name"),
            "email": email,
            "age": user_node.get("age"),
            "gender": user_node.get("gender"),
            "interests": user_node.get("interests", []),
            "profile_picture": user_node.get("profile_picture", "")
        }
    except Exception as e:
        raise HTTPException(status_code=401, detail="Invalid token or internal error")
```

main.py

```
# --- Endpoint para registrarnos ---
@app.post("/register")
def register(user: UserCreate):
    try:
        create_user(driver, user.email, user.password, user.name, user.age, user.gender, user.interests, user.profile_picture or "")
        return {"message": "Usuario creado correctamente"}
    except ValueError as e:
        raise HTTPException(status_code=400, detail=str(e))
    except Exception as e:
        raise HTTPException(status_code=500, detail="Error interno del servidor")
```

## script.js

```
// Función para cargar recomendaciones del user logeado
document.addEventListener("DOMContentLoaded", () => {
  loadMyRecommendations();
});

async function loadMyRecommendations() {
  try {
    const response = await fetch("/user-logged-recommendations", {
      method: "GET",
      credentials: "include"
    });

    if (!response.ok) {
      console.error("Error cargando recomendaciones", await response.text());
      return;
    }

    const data = await response.json();
    const list = document.getElementById("my-recommendation-list");
    list.innerHTML = "";

    if (!data.recommendations || data.recommendations.length === 0) {
      list.innerHTML = "<p style='color: white;'>No tienes recomendaciones aún.<br>Agrega a amigos para obtener recomendaciones.</p>";
      return;
    }

    data.recommendations.forEach(person => {
      const card = document.createElement("div");
      card.className = "card";

      if (typeof person === "string") {
        card.textContent = person;
      } else {
        const profileImage = person.profile_picture || "/static/default.jpg";
        const interests = Array.isArray(person.interests) ? person.interests.join(", ") : "No especificados";

        card.innerHTML = `
          
          <h3>${person.name}</h3>
          <p><strong>Edad:</strong> ${person.age}</p>
          <p><strong>Género:</strong> ${person.gender}</p>
          <p><strong>Intereses:</strong> ${interests}</p>
          <p><strong>Motivo:</strong> ${person.reason}</p>
        `;

        list.appendChild(card);
      }
    });
  } catch (error) {
    console.error("Error en la petición a /user-logged-recommendations:", error);
  }
}
```

## profile.html

```
<body>
  <div class="container">
    <h1> Mi Perfil</h1>
    <form id="edit-form">

      <label for="user-profile-picture">Foto de perfil:</label>
      <div style="text-align: center; margin: 15px 0;">
        
      </div>
      <input type="file" id="user-profile-picture" accept="image/*" />

      <label for="user-name">Nombre:</label>
      <input type="text" id="user-name" required />

      <label for="user-email">Email:</label>
      <input type="email" id="user-email" disabled />

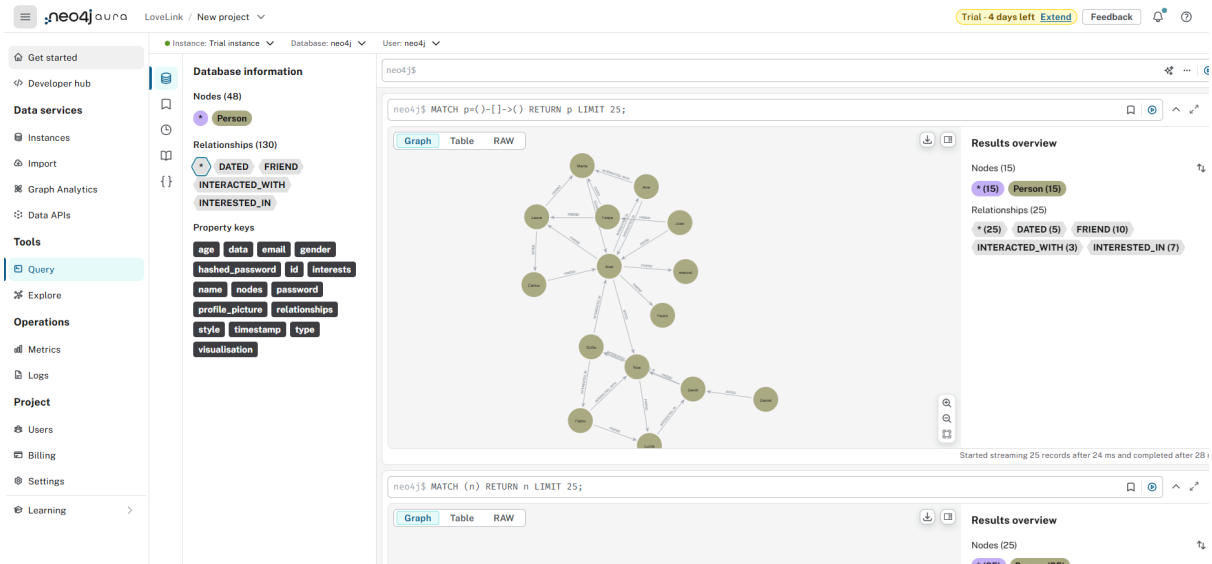
      <label for="user-age">Edad:</label>
      <input type="number" id="user-age" required />
    </form>
  </div>
```



# FASE 8: Despliegue en la nube y en internet.

## 1. Base de datos en la nube

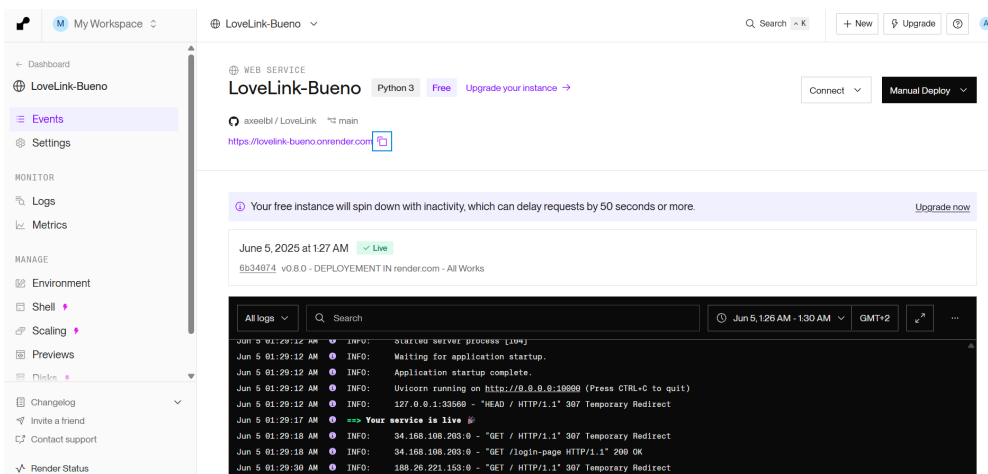
Desde el principio del proyecto la base de datos ya era en la nube en Neo4j Aura.



## 2. Despliegue en Internet.

Una vez que la base de datos ya estaba en la nube, solo nos quedaba hacer accesible la API REST y el Cliente Web desde todo el mundo.

Para eso lo que hemos hecho es subir nuestro proyecto desde github a [Render.com](https://lovelink-bueno.onrender.com) para que esta pagina se encargue del hosting y de la salida a internet automáticamente.



Podéis acceder a la página web desde: <https://lovelink-bueno.onrender.com>

## Estructura Final del Proyecto

LoveLink/

