A continuación, defino una estructura y definiciones para el proyecto- desafío talatask.

**1. Estructura de la API**

La API deberá manejar las siguientes entidades:

* **Empleados**: Nombre, conjunto de habilidades, disponibilidad diaria (en horas), y días disponibles para trabajar(fechas).
* **Tareas**: Título, fecha, duración estimada (en horas), y las habilidades requeridas.

**2. Tecnologías**

* **Lenguaje**: Python con **FastAPI** (rápido para crear APIs RESTful).
* **Almacenamiento**: Utilizaré base de datos en memoria como **SQLite**.
* **Dockerización**: Docker y docker-compose para definir la configuración del entorno.

**3. Arquitectura del Proyecto**

/Proyecto\_Talana

│─── app

│ ├── main.py # Punto de entrada de la API

│ ├── models.py # Definición de los modelos (Empleado, Tarea)

│ ├── database.py # Manejo de la base de datos

│ ├── schemas.py # Validaciones y serialización de datos (pydantic)

│ ├── services.py # Lógica de negocio, como asignación de tareas

│ ├── routers # Endpoints divididos por recursos

│ ├── employees.py # Rutas para empleados

│ └── tasks.py # Rutas para tareas

│─── Dockerfile # Definición del contenedor Docker

│─── docker-compose.yml # Configuración de varios servicios (si es necesario)

│─── README.md # Documentación

**4. Algoritmo de asignación de tareas**

**Lógica básica:**

1. **Recoger disponibilidad y habilidades de empleados**.
2. **Recoger las tareas y su duración**.
3. **Asignar tareas**: El algoritmo deberá:
   * Verificar que el empleado esté disponible el día de la tarea.
   * Asegurarse de que el empleado tenga las habilidades requeridas.
   * Distribuir las horas de trabajo de manera que las tareas se completen sin exceder la disponibilidad diaria de cada empleado.

**Esquema de decisión:**

* Ordena a los empleados según su disponibilidad y habilidades coincidentes con la tarea.
* Asigna la tarea al empleado más adecuado.
* Resta la cantidad de horas de trabajo disponibles.

**5. Pseudocódigo del Algoritmo**

Estructura básica del algoritmo:

1. **Filtrar los empleados disponibles** para la fecha específica de la tarea.
2. **Verificar habilidades**: Cada tarea tiene un conjunto de habilidades requeridas, por lo que sólo se asignará a empleados que las posean.
3. **Asignar tareas respetando la disponibilidad horaria** de cada empleado. Si un empleado tiene suficiente tiempo en su jornada, se le asigna la tarea. Si no, se busca otro empleado.

**6. Código del Algoritmo**

**app/services.py**

# Algoritmo para asignar tareas

def assign\_tasks(db: Session):

    employees = db.query(Employee).all()

    tasks = db.query(Task).all()

    #revisando lo que esta en BD

    #print(f"Tareas: {tasks}")

    #print(f"Empleados: {employees}")

    # Crear un diccionario para almacenar las asignaciones

    assignments = {}

    for task in tasks:

        # Filtrar empleados que están disponibles en la fecha de la tarea

        logging.info("Evaluando tarea: %s", task.title)

        available\_employees = [

            # employee for employee in employees

            # if task.date in employee.available\_days

            # and employee.availability\_hours >= task.duration\_hours

            # and has\_required\_skills(employee.skills.split(','), task.required\_skills.split(','))

               employee for employee in employees

            if task.date in employee.available\_days  # Se asume que available\_days es una lista JSON

            and employee.availability\_hours >= task.duration\_hours

            and has\_required\_skills(employee.skills, task.required\_skills)  # No es necesario el split

        ]

        logging.info("Empleados disponibles: %s", available\_employees)

        if available\_employees:

            # Ordenar por disponibilidad (opcional, para asignar al más disponible)

            available\_employees.sort(key=lambda e: e.availability\_hours, reverse=True)

            # Asignar la tarea al primer empleado disponible

            selected\_employee = available\_employees[0]

            # Actualizar la disponibilidad horaria del empleado

            selected\_employee.availability\_hours -= task.duration\_hours

            # Registrar la asignación

            if selected\_employee.name not in assignments:

                assignments[selected\_employee.name] = []

            assignments[selected\_employee.name].append(task.title)

            # Actualizar la base de datos

            db.commit()

    return assignments

**7. Explicación del Código**

**Función has\_required\_skills**

Esta función verifica si un empleado tiene todas las habilidades necesarias para una tarea. Compara las habilidades requeridas por la tarea con las habilidades del empleado.

**Función assign\_tasks**

Este es el núcleo del algoritmo:

* Primero, se obtienen todos los empleados y tareas de la base de datos.
* Luego, para cada tarea:
  1. Se filtran los empleados que estén disponibles en la fecha de la tarea, que tengan suficientes horas y que tengan las habilidades requeridas.
  2. Entre los empleados disponibles, se selecciona el que tiene más horas disponibles (opcional, pero útil para maximizar la eficiencia).
  3. Se asigna la tarea a ese empleado, se actualiza su disponibilidad horaria y se registra la asignación.
* Finalmente, el algoritmo devuelve un diccionario con las asignaciones de tareas por empleado.

**Función get\_task\_assignments**

* Esta función utiliza el algoritmo de asignación y filtra los resultados para una fecha específica.
* Devuelve las asignaciones de tareas por empleado, para que luego se puedan mostrar en los endpoints.

**8. Endpoints**

* **POST /employees**: Crear nuevos empleados.
* **POST /tasks**: Crear nuevas tareas.
* **POST /assignments**: Ejecutar la asignación de tareas.
* **GET /assignments/{fecha}**: Obtener un reporte de las asignaciones de un día específico.

**9. Configuración de SQLite**

Se crea un archivo database.py para manejar la conexión a SQLite usando SQLAlchemy.

**app/database.py**

from sqlalchemy import create\_engine, inspect

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy.orm import sessionmaker

DATABASE\_URL = "sqlite:///./test.db"

engine = create\_engine(DATABASE\_URL, connect\_args={"check\_same\_thread":False})

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative\_base()

def create\_employee\_table():

    from models import Employee

    inspector = inspect(engine)

    existing\_tables = inspector.get\_table\_names()

    if "employees" not in existing\_tables:

        Base.metadata.create\_all(bind=engine, tables=[Employee.\_\_table\_\_])

def create\_task\_table():

    from models import Task

    inspector = inspect(engine)

    existing\_tables = inspector.get\_table\_names()

    if "tasks" not in existing\_tables:

        Base.metadata.create\_all(bind=engine, tables=[Task.\_\_table\_\_])

def get\_db():

    db = SessionLocal()

    try:

        yield db

    finally:

        db.close()

**10. Definición de Modelos**

Definición de los modelos para empleados y tareas en models.py.

**app/models.py**

from sqlalchemy import Column, Integer, String, Boolean, ForeignKey, Table

from sqlalchemy.orm import relationship

from database import Base

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy.types import TypeDecorator, VARCHAR

import json

# Se define un tipo personalizado para serializar listas como JSON

class JsonType(TypeDecorator):

    impl = VARCHAR

    def process\_bind\_param(self, value, dialect):

        if value is not None:

            return json.dumps(value)

        return value

    def process\_result\_value(self, value, dialect):

        if value is not None:

            return json.loads(value)

        return value

class Employee(Base):

    \_\_tablename\_\_ = "employees"

    id = Column(Integer, primary\_key=True, index=True)

    name = Column(String, index=True)

    skills = Column(JsonType)

    availability\_hours = Column(Integer)

    available\_days = Column(JsonType)

class Task(Base):

    \_\_tablename\_\_ = "tasks"

    id = Column(Integer, primary\_key=True, index=True)

    title = Column(String, index=True)

    required\_skills = Column(JsonType)

    duration\_hours = Column(Integer)

    date = Column(String)

**11. Dockerización**

* **Dockerfile**:
  + Contendrá instrucciones para levantar la API con Python y FastAPI.
* # Usa una imagen de Python ligera
* FROM python:3.9-slim
* # Establece el directorio de trabajo en el contenedor
* WORKDIR /app
* # Copia los requerimientos y los instala
* COPY requirements.txt .
* RUN pip install --no-cache-dir -r requirements.txt
* # Copia el resto del código de la aplicación
* COPY . .
* # Expone el puerto que utilizará FastAPI
* EXPOSE 8000
* # Comando para iniciar la aplicación
* CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
* **docker-compose.yml**

Aunque no es estrictamente necesario con SQLite, un docker-compose.yml puede facilitar la ejecución de servicios adicionales si decides agregar más adelante. Pero con SQLite, la ventaja es que no necesitas un servicio de base de datos separado.

* version: '3'
* services:
* api:
* build: .
* ports:
* - "8000:8000"
* volumes:
* - .:/app

**12. Inicialización de la Base de Datos**

Cuando se levante el contenedor, se deben crear las tablas en SQLite.

**app/main.py**

from fastapi import FastAPI

from database import engine, create\_employee\_table, create\_task\_table

from models import Employee, Task

from database import get\_db

from routers import employees, tasks

from sqlalchemy import inspect

# Inicializa la app FastAPI

app = FastAPI()

# Crear las tablas

# models.Base es creado por SQLAlchemy y contiene la definición de las tablas

#models.Base.metadata.create\_all(bind=engine)

create\_employee\_table()

create\_task\_table()

# Incluir routers / endpoints

app.include\_router(employees.router)  # Endpoints relacionados con empleados

app.include\_router(tasks.router)      # Endpoints relacionados con tareas

# Endpoint raíz

# http://127.0.0.1:8000/

@app.get("/")

def read\_root():

    return {"message": "Bienvenido a la API Asignadora de tareas"}

def list\_tables():

    inspector = inspect(engine)

    tables = inspector.get\_table\_names()

    print("Tablas:", tables)

list\_tables()

**13. Comandos para Dockerizar y Ejecutar**

* **Construir el contenedor:**

docker build -t task-api .

* **Correr el contenedor:**

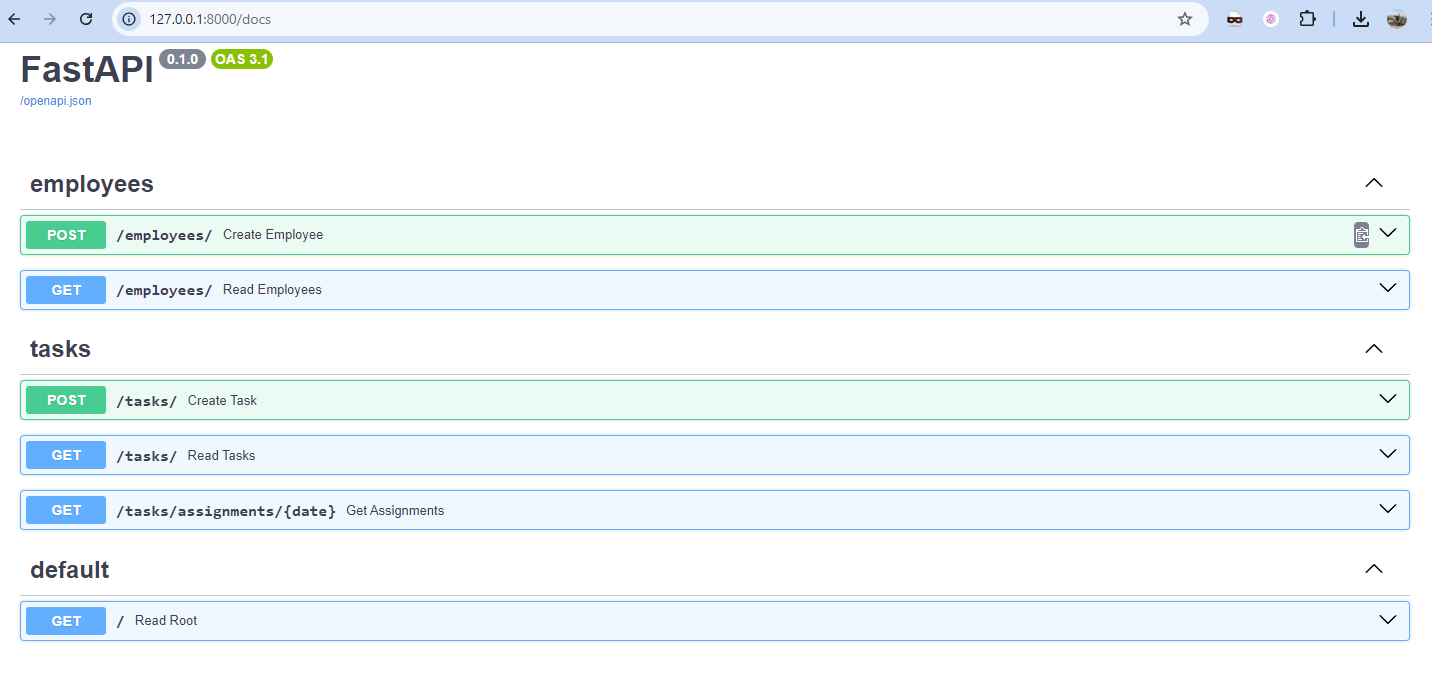
docker run -d -p 8000:8000 task-api

Este setup permitirá usar SQLite dentro de Docker. Dado que SQLite crea un archivo en el sistema de archivos, el contenedor podrá usar ese archivo sin problemas. Además, la API estará completamente encapsulada en Docker, lo que facilita la portabilidad.

**14. Pruebas**

* Usare un archivo en Python que simulara una llamada a los endpoints y al algoritmo de asignación, para asegurarme de que las asignaciones se realicen correctamente y que los endpoints funcionen como se espera.
* Para probar la API, se utilizó el explorador de API, que viene integrado en FastAPI, este se encuentra disponible en: <http://localhost:8000/docs>.

A continuación, una imagen de cómo se ve el explorador de FastAPI:



**15. Documentación**

* Es esencial incluir una **guía de instalación** y **uso** en el README.md para que se pueda levantar el proyecto fácilmente.

**16. Instalación de FastAPI**

Para instalar FastApi se deben ejecutar las siguientes instrucciones en CMD, que está apuntando a la carpeta donde se guardó el proyecto.

**17. Instalación de Uvicorn**

Para instalar Uvicorn se deben ejecutar las siguientes instrucciones en CMD, que está apuntando a la carpeta donde se guardó el proyecto.

**18. Instalación de ambiente, para trabajar con FastAPI y Python**

Lo primero que se debe hacer es crear un ambiente virtual para trabajar con este proyecto, utilizando los siguientes comandos, abriendo una terminal CMD, que este apuntando a la carpeta física donde alojaremos el proyecto:

Instrucciones sacadas de la documentación oficial de Python desde <https://docs.python.org/es/3/tutorial/venv.html>

Para crear un entorno virtual, decide en que carpeta quieres crearlo y ejecuta el módulo [venv](https://docs.python.org/es/3/library/venv.html" \l "module-venv" \o "venv: Creation of virtual environments.) como script con la ruta a la carpeta:

python -m venv tutorial-env

Esto creará el directorio tutorial-env si no existe, y también creará directorios dentro de él que contienen una copia del intérprete de Python y varios archivos de soporte.

Una ruta común para el directorio de un entorno virtual es .venv. Ese nombre mantiene el directorio típicamente escondido en la consola y fuera de vista mientras le da un nombre que explica cuál es el motivo de su existencia. También permite que no haya conflicto con los ficheros de definición de variables de entorno .env que algunas herramientas soportan.

Una vez creado el entorno virtual, podrás activarlo.

En Windows, ejecuta:

tutorial-env\Scripts\activate

A continuación, dentro de este ambiente virtual, instalaremos las dependencias necesarias, para que nuestro proyecto funcione con las tecnologías que mencionamos anteriormente:

FastAPI:

pip install fastapi

Uvicorn:

pip install uvicorn

luego de esto se debería poder levantar el servidor, con el comando:

uvicorn main:app –reload.