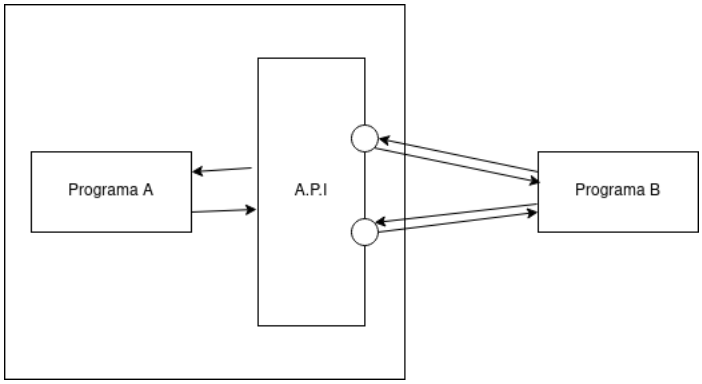
Tutorial 1: Diseñar, Documentar, Implementar y Despliegar una API web con el framework FastAPI

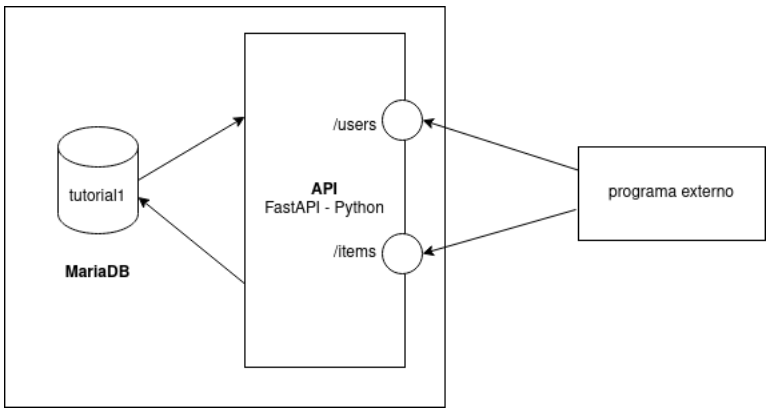


Una API (*application programming interface*) permite la comunicación entre dos programas. El programa A expone metodos o “endpoints” que el programa B puede utilizar, ocultando la complejidad interna del programa A.

1. Requisitos no funcionales

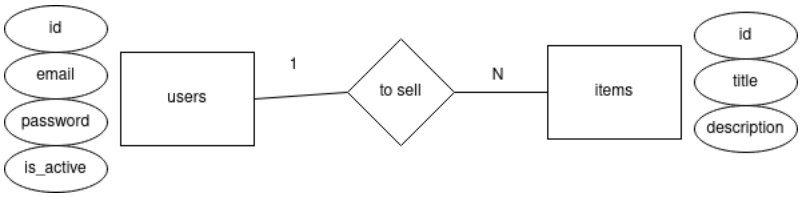
- **Compatibilidad/Interoperabilidad**: Capacidad de dos o más sistemas o componentes para intercambiar información y utilizar la información intercambiada.
- **Mantenibilidad/Capacidad de ser modificado**: Capacidad del producto que permite que sea modificado de forma efectiva y eficiente sin introducir defectos o degradar el desempeño. Se puede modificar el programa A, sin afectar el programa B, a condición que la API no cambie.

2. Contexto del tutorial



Queremos exponer datos de una base de datos, llamada "tutorial1", a través una API web.

La base de datos "tutorial" tiene el modelo Entidad-Relación siguiente:



Modelo Entidad-Relación de la base de datos tutorial1.

Nos gustaría exponer algunos endpoints, via una API web, que permitirán a programas externos de comunicar con la base de datos según ciertos escenarios.

3. Diseño de la API Web

Definir los recursos, verbos, parámetros y respuestas:

3.1 /users/

- `GET` : leer usuarios

Parámetros:

1. skip (int): pasar los primeros elementos
2. limit (int): numero máximo de respuestas

Ejemplo de consulta:

`http://127.0.0.1:8000/users/?skip=0&limit=100`

Ejemplo de respuesta:

```
[
  {
    "email": "string",
    "id": 0,
    "is_active": true,
    "items": []
  }
]
```

- `POST` : subir un nuevo usuario

Sin parámetro.

3.2 /users/{user_id}/

- `GET` : leer la información de un usuario específico identificado por su id

Sin parámetros.

3.3 /users/{user_id}/items

- `POST` : subir un nuevo item para un usuario específico

Sin parámetros.

3.4 /items/

- `GET` : leer la lista de items

Parámetros:

1. skip (int): pasar los primeros elementos
2. limit (int): numero máximo de respuestas

Ejemplo de consulta:

`http://127.0.0.1:8000/items/?skip=0&limit=100`

Ejemplo de respuesta:

```
[
  {
    "title": "string",
    "description": "string",
    "id": 0,
    "owner_id": 0
  }
]
```

Algunas buenas prácticas para diseñar una API:

💡 Evitar poner el verbo HTTP en la ruta

💡 Anticipar que pueden haber futuras versiones de la API:

```
/v1/users?skip=0&limit=100
```

4. Implementar y Documentar una API web con FastAPI

> Esta sección se inspira de https://fastapi.tiangolo.com/tutorial que pueden revisar para tener más detalles.

FastAPI es un framework web moderno y de alto rendimiento para construir APIs con Python: https://fastapi.tiangolo.com/

```
$ pip install fastapi
$ pip install uvicorn
```

Uvicorn es el servidor que utilizará la API para servir peticiones.

4.1 API web básica para recibir consultas GET

- Crear una primera API básica:

```
# main.py

from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def root():
    return {"message": "Hello World"}
```

- Ejecutar la API via uvicorn:

```
$ uvicorn main:app --reload

INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)ImportError: attempted relative import with no known parent packag
```

💡 ver la API: http://127.0.0.1:8000/

💡 ver la documentación interactiva de la API: http://127.0.0.1:8000/docs

💡 ver la documentación alternativa: http://127.0.0.1:8000/redoc

- Definir rutas y parámetros de ruta:

```
# main2.py

from fastapi import FastAPI

app = FastAPI()

@app.get("/items/{item_id}")
async def read_item(item_id: int):
    return {"item_id": item_id}
```

- Definir rutas y **parámetros de consulta**:

```
# main3.py

from fastapi import FastAPI

app = FastAPI()

fake_items_db = [{"item_name": "Foo"}, {"item_name": "Bar"}, {"item_name": "Baz"}]

@app.get("/items/")
async def read_item(skip: int = 0, limit: int = 10):
    return fake_items_db[skip : skip + limit]
```

💡 ver http://127.0.0.1:8000/items/?skip=0&limit=10

- Parámetros opcionales:

```
#main4.py

from typing import Optional

from fastapi import FastAPI

app = FastAPI()


@app.get("/items/{item_id}")
async def read_item(item_id: str, q: Optional[str] = None):
    if q:
        return {"item_id": item_id, "q": q}
    return {"item_id": item_id}
```

💡 ver http://127.0.0.1:8000/items/3?q=test

- Multiples parámetros de ruta y consulta:

```
#main5.py

from typing import Optional

from fastapi import FastAPI

app = FastAPI()


@app.get("/users/{user_id}/items/{item_id}")
async def read_user_item(
    user_id: int, item_id: str, q: Optional[str] = None, short: bool = False
):
    item = {"item_id": item_id, "owner_id": user_id}
    if q:
        item.update({"q": q})
    if not short:
        item.update(
            {"description": "This is an amazing item that has a long description"}
        )
    return item
```

💡 ver http://127.0.0.1:8000/users/3/items/hola?q=test&short=False

- Parámetros obligatorios:

```
#main6.py

from fastapi import FastAPI

app = FastAPI()

@app.get("/items/{item_id}")
async def read_user_item(item_id: str, needy: str):
    item = {"item_id": item_id, "needy": needy}
    return item
```

💡 ver http://127.0.0.1:8000/items/hola

4.2 API Web básica para recibir consultas POST

Para la consultas de tipo POST, el cliente (navegador, programa externo, etc.) envia datos a través el cuerpo de su consulta (**Request Body**).

Para gestionar los Request Body, FastAPI utiliza el paquete `pydantic`

```
#main7.py

from typing import Optional

from fastapi import FastAPI
from pydantic import BaseModel


class Item(BaseModel):
    name: str
    description: Optional[str] = None
```

```
    price: float
    tax: Optional[float] = None


app = FastAPI()


@app.post("/items/")
async def create_item(item: Item):
    return item
```

ejemplos de Request Body:

```
{
    "name": "Foo",
    "description": "An optional description",
    "price": 45.2,
    "tax": 3.5
}

{
    "name": "Foo",
    "price": 45.2
}
```

- Utiizar los datos enviados por el cliente:

```
#main8.py

from typing import Optional

from fastapi import FastAPI
from pydantic import BaseModel


class Item(BaseModel):
    name: str
    description: Optional[str] = None
    price: float
    tax: Optional[float] = None


app = FastAPI()

@app.post("/items/")
async def create_item(item: Item):
    item_dict = item.dict()
    if item.tax:
        price_with_tax = item.price + item.tax
        item_dict.update({"price_with_tax": price_with_tax})
    return item_dict
```

- Combinar el Body Request con parámetros de ruta y de consulta:

```
#main9.py
from typing import Optional

from fastapi import FastAPI
from pydantic import BaseModel


class Item(BaseModel):
    name: str
    description: Optional[str] = None
    price: float
    tax: Optional[float] = None


app = FastAPI()


@app.put("/items/{item_id}")
```

```
async def create_item(item_id: int, item: Item, q: Optional[str] = None):
    result = {"item_id": item_id, **item.dict()}
    if q:
        result.update({"q": q})
    return result
```

5. Conectar la API Web a una base de datos relacional

> Esta sección se inspira de https://fastapi.tiangolo.com/tutorial/sql-databases/ que pueden revisar para tener más detalles.

En esta sección, creamos una carpeta con los archivos para nuesta aplicación:

```
.
└── my-api
    ├── __init__.py         #un archivo vacio --> la carpeta es un package
    ├── crud.py
    ├── database.py
    ├── main.py
    ├── models.py
    └── schemas.py
```

- Crear un usuario para acceder a MariaDB:

```
$ sudo mysql -u root -p
```

```
CREATE USER 'my-api'@'localhost' IDENTIFIED BY 'my-api-password';
GRANT ALL PRIVILEGES ON * . * TO 'my-api'@'localhost';
```

- **database.py**: conexión al SGBD MariaDB

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

SQLALCHEMY_DATABASE_URL = "mysql+mysqlconnector://my-api:my-api-password@127.0.0.1:3306/tutorial1"

engine = create_engine(SQLALCHEMY_DATABASE_URL)

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

Base = declarative_base()
```

`SessionLocal` es el objeto que representa la conexión a la base de datos.

`Base` es el objeto que representa el modelo de la base de datos.

- **models.py**: representación del modelo de la base de datos por SQLAlchemy

```
from sqlalchemy import Boolean, Column, ForeignKey, Integer, String
from sqlalchemy.orm import relationship

from .database import Base #Se importa el objeto Base desde el archivo database.py

class User(Base):

    __tablename__ = "users"

    id = Column(Integer, primary_key=True, index=True)
```

```
    email = Column(String(50), unique=True, index=True)
    hashed_password = Column(String(50))
    is_active = Column(Boolean, default=True)

    items = relationship("Item", back_populates="owner")

class Item(Base):

    __tablename__ = "items"

    id = Column(Integer, primary_key=True, index=True)
    title = Column(String(50), index=True)
    description = Column(String(100), index=True)
    owner_id = Column(Integer, ForeignKey("users.id"))

    owner = relationship("User", back_populates="items")
```

El atributo `__tablename__` indica a SQLalchemy el nombre de la tabla que utilizar en MariaDB.

- **schema.py**: representación de los recursos de la API por pydantic

```
from typing import List, Optional

from pydantic import BaseModel


class ItemBase(BaseModel):
    title: str
    description: Optional[str] = None


class ItemCreate(ItemBase):
    pass


class Item(ItemBase):
    id: int
    owner_id: int

    class Config:
        orm_mode = True


class UserBase(BaseModel):
    email: str


class UserCreate(UserBase):
    password: str


class User(UserBase):
    id: int
    is_active: bool
    items: List[Item] = []

    class Config:
        orm_mode = True
```

- **crud.py:** lectura y escritura a la base de datos

```
from sqlalchemy.orm import Session

from . import models, schemas


def get_user(db: Session, user_id: int):
    return db.query(models.User).filter(models.User.id == user_id).first()


def get_user_by_email(db: Session, email: str):
    return db.query(models.User).filter(models.User.email == email).first()
```

```
def get_users(db: Session, skip: int = 0, limit: int = 100):
    return db.query(mouser_iddels.User).offset(skip).limit(limit).all()


def create_user(db: Session, user: schemas.UserCreate):
    fake_hashed_password = user.password + "notreallyhashed"
    db_user = models.User(email=user.email, hashed_password=fake_hashed_password)
    db.add(db_user)
    db.commit()
    db.refresh(db_user)
    return db_user


def get_items(db: Session, skip: int = 0, limit: int = 100):
    return db.query(models.Item).offset(skip).limit(limit).all()


def create_user_item(db: Session, item: schemas.ItemCreate, user_id: int):
    db_item = models.Item(**item.dict(), owner_id=user_id)
    db.add(db_item)
    db.commit()
    db.refresh(db_item)
    return db_item
```

- **main.py**: la API Web FastAPI con sus endpoints

```
from typing import List

from fastapi import Depends, FastAPI, HTTPException
from sqlalchemy.orm import Session

from . import crud, models, schemas
from .database import SessionLocal, engine

models.Base.metadata.create_all(bind=engine)

app = FastAPI()


# Dependency
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()


@app.post("/users/", response_model=schemas.User)
def create_user(user: schemas.UserCreate, db: Session = Depends(get_db)):
    db_user = crud.get_user_by_email(db, email=user.email)
    if db_user:
        raise HTTPException(status_code=400, detail="Email already registered")
    return crud.create_user(db=db, user=user)


@app.get("/users/", response_model=List[schemas.User])
def read_users(skip: int = 0, limit: int = 100, db: Session = Depends(get_db)):
    users = crud.get_users(db, skip=skip, limit=limit)
    return users


@app.get("/users/{user_id}", response_model=schemas.User)
def read_user(user_id: int, db: Session = Depends(get_db)):
    db_user = crud.get_user(db, user_id=user_id)
    if db_user is None:
        raise HTTPException(status_code=404, detail="User not found")
    return db_user


@app.post("/users/{user_id}/items/", response_model=schemas.Item)
def create_item_for_user(
    user_id: int, item: schemas.ItemCreate, db: Session = Depends(get_db)
):
    return crud.create_user_item(db=db, item=item, user_id=user_id)
```

```
@app.get("/items/", response_model=List[schemas.Item])
def read_items(skip: int = 0, limit: int = 100, db: Session = Depends(get_db)):
    items = crud.get_items(db, skip=skip, limit=limit)
    return items
```

Ejecutar la aplicación con:

$ `uvicorn my-api.main:app --reload`

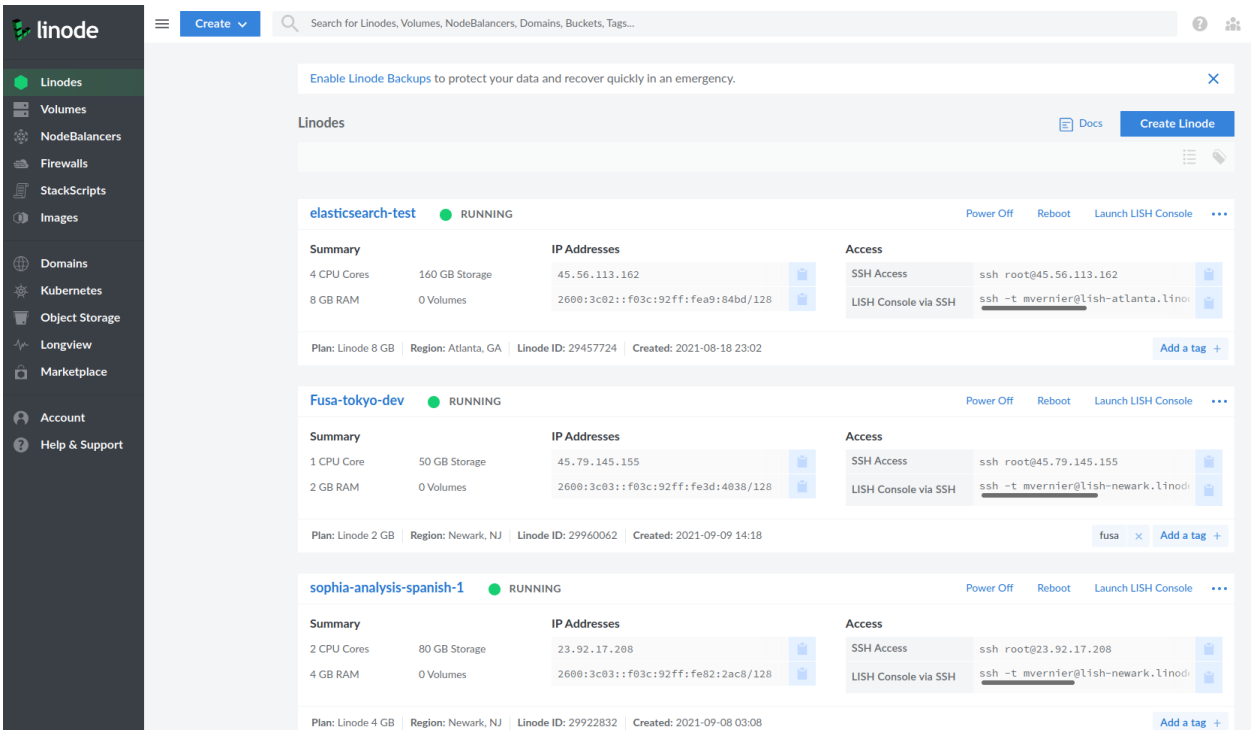
💡 ver http://127.0.0.1:8000/docs

💡 ver su instancia de MariaDB en el terminal.

6. Desplegar (manualmente) la base de datos y API en un servidor

- Arrendar un servidor

  por ejemplo en Linode.com



- Conectarse a un servidor por SSH:

`ssh root@172.104.24.202`

- Clonear el repositorio:

`cd /home`

`git clone https://github.com/matthieuvernier/INFO229_2021`

- Instalar las dependencias:

```
sudo apt update
```

```
sudo apt install python3-pip

pip install uvicorn

pip install fastapi

pip install sqlalchemy

pip install mysql-connector-python
```

- Instalar MariaDB y crear el usuario my-api

```
sudo apt install mariadb-server
```

```
  CREATE USER 'my-api'@'localhost' IDENTIFIED BY 'my-api-password';
GRANT ALL PRIVILEGES ON * . * TO 'my-api'@'localhost';

CREATE DATABASE tutorial1;
```

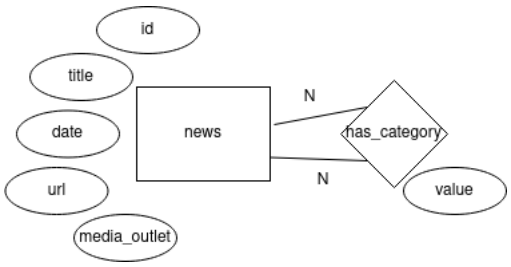
- Depliegar la API:

`uvicorn my-api.main:app --host 0.0.0.0 --port 80`

💡 ¿Qué opinan de la Portabilidad / Despliegabilidad de nuestra API?

7. Ejercicio

Supongamos que tenemos la base de datos siguiente:



Esta base de datos, llamada *Sun*, almacena información sobre noticias de prensa: el título de la noticia, su fecha de publicación, su url y el nombre de medio que publicó la noticia. Además una noticia puede pertenecer a 1 o varias categorías: política, deporte, economia, ecología, salud, etc.

1. Crear la base de datos en MariaDB agregando unos datos ficticios en las tablas "news" y "has_category"

Base de datos creada y adjunta en GitHub

1. Diseñar e implementar una API web para servir el *endpoint* siguiente:
   - GET /v1/news?from=2021-01-01&to=2021-01-31&category=sport

   Resultado esperado:

```
[
  {
    "id": int,
    "title": "string",
    "url": "string",
```

```
    "date": "string",
    "media_outlet": "string",
    "category": "string"
  }
]
```

No pude concluir main. de todas maneras adjunté mi trabajo en GitHub, puede revisar los otros archivos