Integración Web de BDD

TB022 - Esteban - Riesgo - Kristal

ORM

*Object Relational Mapper

Según el RDBMS que usemos la integración será distinta aunque bastante parecida.

El setup incluye: manejar la conexión durante el inicio del servidor, definir los modelos que serán nuestras tablas, y las *queries* que queramos hacer.

Esta integración se suele hacer a través de un ORM.

Un ORM es una biblioteca que se encarga justamente de manejar todo lo que tiene que ver con bases de datos en nuestro sistema.

Como siempre, hay muchas opciones para elegir, pero en nuestro caso usaremos SQLModel.

SQLModel

Por qué?

- Fácil de usar y comprender
- Basado en Pydantic y en SQLAlchemy (el ORM más popular) con algunas mejoras.
- Pensado para ser usado junto con FastAPI
- Hecho por el mismo creador de FastAPI
- No se usa SQL directo, sino que se usa una interfaz propia par interactuar con la DB

Instalación

```
pip install sqlmodel
```

Conectar una DB a una aplicación

En el caso de SQLModel (y en la mayoría de los ORM) generar la conexión entre una aplicación y una DB es muy sencillo, se quieren los siguientes pasos:

Definir la conexión

Hacer el set up con la DB a la cual querramos conectarnos: definir el nombre, credenciales a usar y URL (si fuese necesario), modo de conexión y algunas otras cosas.

Definir el schema

Definir qué versión de la DB vamos a usar. Generalmente se define mediante el uso de *migraciones*.

Migraciones

Como habiamos adelantado, manejar una base de datos en un ambiente real es *complicado*.

Si hacemos cambios a la DB, pero hay otros sistemas usandola activamente (por ejemplo, otros colaboradores del equipo) es necesario manejar también versiones del schema de la base de datos.

El schema define la forma de la DB (qué tablas hay, qué columnas tiene cada una y con qué constraints). Cada vez que una sentencia modifique una tabla o un columna de una tabla, se modificará el schema.

Si queremos utilizar una DB en nuestro sistema, y el sistema tiene versiones, es necesario entonces versionar también la DB.

Para ello se usan *migraciones*

Migraciones (DB migrations)

Una migración es simplemente un script que contiene los cambios a aplicar sobre una DB para pasar de una version N a la version N+1 (la siguiente).

El historial de migraciones es lineal, encadenada. Toda migración tiene exactamente una migración anterior (excepto la primera, la inicial) y exactamente una migración siguiente (excepto la última).

Estos cambios pueden ser sobre el schema (tablas y columnas y constraints) o también sobre los datos mismos.

Este script puede estar escrito en SQL directo o utilizar herramientas de más alto nivel que escribirán el SQL por nosotros.

Nosotros vamos a usar Alembic.

1. Establecer conexión

Para conectar nuestra aplicación con la DB necesitamos crear un "engine":

```python
from sqlmodel import create_engine

root = os.path.abspath(os.path.join(os.path.dirname(__file__), ".."))
SQLITE_FILE_PATH = os.path.join(root, "database.db")

engine = None

def init_engine():
    global engine
    engine = create_engine(f"sqlite:///{SQLITE_FILE_PATH}", echo=True)
```

Vamos a tener que usar este engine desde nuestra clase Repository/Database que teníamos previamente.

Es importante recordar que el path a la DB debería ser en algún lugar que tenga sentido (dentro del directorio del repo) pero **NO DEBEMOS PUSHEAR** el archivo de la DB al remoto.

Este ejemplo busca el archivo database.db en el directorio root del repositorio.

2. Crear una migración

Este paso si bien no es necesario per se, si no creamos nada la DB va a estar vacía y no va a servir de mucho.

Por lo tanto, vamos a crear una primer migración que cree la tabla de `Alumno` con los datos que veníamos usando.

Primero, por única vez, ejecutamos el setup de la herramienta `alembic` que vamos a usar para las migraciones:

```
$ alembic init alembic
```

Nos creará un directorio `/alembic` en el root del repositorio, ahí se guardarán las migraciones y archivos de configuración. Debemos modificar el archivo `alembic.ini` para que apunte a la DB que queremos:

```
sqlalchemy.url = sqlite:///<path/to/db>
```

y descomentar la **línea 12**:

```
file_template = %%(year)d_%%(month).2d_%%(da....
```

Creemos la primer migración ahora con:

```
$ alembic revision -m "Crear tabla alumnos"
```

Lo cual crea un archivo en `/alembic/revisions/` donde vamos a poner la lógica de la migración

2. Crear una migración: contenido

Una migración tiene 2 partes:

`upgrade`

Se ejecutará al correr la migración normalmente. Se hace un "upgrade" de la versión de la DB que incluye la versión de la migración actual.

`downgrade`

Proceso inverso. Se pueden deshacer los cambios hechos por una migración. Se hace un "downgrade" de la version de la DB, que excluye la versión de la migración actual para incluir hasta la de la migración anterior.

De esta manera podemos modificar nuestra DB para que apunte a la versión que querramos.

2. Crear una migración: contenido

```
from alembic import op
def upgrade() -> None:
   op.create_table(
       'alumno',
       sa.Column('padron', sa.Integer, primary_key=True),
       sa.Column('nombre', sa.Text, nullable=False),
       sa.Column('apellido', sa.Text, nullable=False),
       sa.Column('edad', sa.Integer)
   )
```

La acción opuesta de crear una tabla es destruirla

```
 def downgrade() -> None:
     op.drop_table('alumno')
```

3. Correr todas las migraciones pendientes

Es recomendable chequear si hay migraciones pendientes por correr cada vez que se pullee código hecho por alguien más (o cuando nosotros mismos las agregamos).

```
$ alembic upgrade head
```

Ejecutará la sección de `upgrade()` de todas las migraciones las cuales no se hayan corrido aún. La herramienta sabe detectar cuál es la migración aplicada más reciente según la versión del schema.

4. Crear el modelo Alumno

Ahora que ya creamos la tabla, debemos crear un modelo en Python que la represente. Podemos reusar el modelo que ya teníamos haciendo algunas modificaciones:

```python
class AlumnoBase(SQLModel):
    nombre: str
    apellido: str
    edad: int | None = Field(default=None, ge=17)

class Alumno(AlumnoBase, table=True):
    padron: int = Field(primary_key=True)

class AlumnoUpsert(AlumnoBase):
    pass
```

Podemos agregar validaciones sobre los campos

No se olviden de poner que esta clase representa a una tabla en la DB

Esto es para que sea igual que la clase base, para mantener el nombre que veníamos usando

5. Actualizar la lógica del proyecto

Nuestra clase Database ahora va a conectarse con la RDB de Sqlite que estuvimos configurando. Para ello necesitamos configurar dos cosas: engine y sessions. El primero ya lo tenemos, nos falta lo segundo.

Una session nos va a servir para anotar todas las queries que vayamos a querer ejecutar antes de que efectivamente se ejecuten.

En una misma *session* podemos establecer la cantidad de `inserts/updates/deletes/select` que querramos y sólo se ejecutarán al hacer un commit/exec. Si llega a haber algún error en el código entre las queries y el commit ninguna tendrá efecto y no se verán reflejados los cambios en la base de datos.

Esto generalmente es conocido como una *transacción*.

Normalmente el manejo de sessions lo incluiríamos únicamente dentro de la clase Database para manejarlo internamente.

Lamentablemente no podemos hacer eso en nuestro caso, vamos a necesitarlas en los endpoints también, pero podemos manejarlas como dependencias y pasarselas a la DB en cada método.

5. Actualizar la lógica del proyecto

Creamos una nueva dependencia para las sesiones

```python
def get_session() -> Generator[Session, None, None]:
    with Session(engine) as session:
        yield session


SessionDep = Annotated[Session, Depends(get_session)]
```

Actualizamos nuestros endpoints para recibir una session y pasársela a la DB

```python
@router.get("/")
def list(session: SessionDep, db: DatabaseDep) -> list[Alumno]:
   return db.list(session)
```

Cada request al BE tendrá su propia session.

5. Actualizar la lógica del proyecto

Ahora sí, cómo sería un método de la clase Database que consulte la lista de alumnos:

```python
def list(self, session: Session) -> list[Alumno]:
    query = select(Alumno)
    alumnos = session.exec(query).all()
    return alumnos
```

1. Recibimos una nueva session para ejecutar nuestro select.

2. Armamos la query que queremos

3. Ejecutamos la query y decimos que queremos listar todas las filas de la tabla, sin ninguna condición.

Búsquedas: filtros

Las búsquedas nos suelen permitir filtrar los resultados que queremos según determinados criterios

Podríamos querer filtrar nuestros resultados de Alumnos por nombre, o edad, o nombre y apellido, etc.

Simplemente debemos agregar los `where` necesarios a nuestra query SQL. Pero para eso necesitamos hacerle llegar los datos que nos manda el cliente primero.

Como la request es un `GET` necesitamos mandar los filtros por query params. Podemos hacerlo de manera individual cada uno, o si tenemos muchos enviar todo en un único objeto.

Búsquedas: filtros

Agregamos el modelo

```python
class FiltrosAlumno(SQLModel):
    nombre: str | None = None
    apellido: str | None = None
    edad: int | None = None
    padron: int | None = None
```

Agregamos la lógica a la DB

```python
def __build_list_query(self, filtros: FiltrosAlumno):
    query = select(Alumno)
    if filtros:
        for key, val in filtros.model_dump(exclude_none=True).items():
            if key in ["nombre", "apellido"]:
                query = query.where(Alumno.nombre.like(f"%{val}%"))
            else:
                query = query.where(getattr(Alumno, key) == val)
    return query
```

Agregamos el argumento al endpoint

```python
filtros: Annotated[FiltrosAlumno, Query()]
```

```python
@router.get("/")
def list(session: SessionDep, db: DatabaseDep, filtros: Annotated[FiltrosAlumno, Query()]) -> list[Alumno]:
   return db.list(session, filtros)
```

Búsquedas: paginación

No vamos a querer siempre devolver todos los elementos de una tabla, de hecho, pocas veces realmente pasa esto, generalmente queremos buscar con algún filtro.

Y aunque no haya un filtro activo, buscar y devolver todos los elementos no es recomendable. Si nuestra tabla es muy grande podemos llegar a tener problemas de performance.

Lo que vamos a querer hacer es *paginar* nuestros resultados.

Nuestros resultados van a estar acomodados en páginas de cierta cantidad, y el cliente le pedirá al servidor backend los resultados de la página que desee.

Hay muchas formas de lograr este resultado, veamos la más simple: `limit` y `offset`.

Búsquedas: limit & offset

`limit`: limitar la cantidad de resultados a cierto número

`offset`: no incluir los primeros <N> resultados

Usando estos dos parámetros podemos definir páginas a nuestro resultado. El cliente es el encargado de mantener la cuenta y usar los valores correctos para estos dos parámetros.

Si el cliente no especifica estos valores, se usarán valores default.

Los mismos los usaremos a nivel DB, es decir, van a ser parte de la query SQL que hagamos.

Si queremos devolver resultados en páginas de 20 elementos,
la primer página tendra `limit 20` y `offset 0`
la segunda, `limit 20` y `offset 20`
la tercera, `limit 20` y `offset 40`

Si queremos devolver resultados en páginas de N elementos,
la primer página tendrá `limit N` y `offset 0`
la segunda, `limit N` y `offset N`
la K-ésima página tendrá `limit N` y `offset N * (K-1)`

Búsquedas: limit & offset

Agregamos los campos al modelo

```
class FiltrosAlumno(SQLModel):
    ...
    limit: int = Field(50, gt=0, le=50)
    offset: int = Field(0, ge=0)
```

Agregamos la lógica a la DB

```
def __build_list_query(self, filtros: FiltrosAlumno):
    query = select(Alumno)
    if filtros:
        for key, val in filtros.model_dump(exclude=["limit", "offset"],
exclude_none=True).items():
            if key in ["nombre", "apellido"]:
                query = query.where(Alumno.nombre.like(f"%{val}%"))
            else:
                query = query.where(getattr(Alumno, key) == val)

    query = query.limit(filters.limit).offset(filters.offset)
    return query
```

Find

Hacemos los mismos cambios en la DB que en el método anterior

```python
def find(self, session: Session, padron: int) -> Alumno:
    alumno = session.exec(select(Alumno).where(Alumno.padron == padron)).first()
    if alumno:
        return alumno
    raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Alumno not found")
```

Buscamos por padrón

Devuelve el primero o None

Add

Mismos pasos que hasta ahora. En este caso vamos a dejar que la DB le asigne automáticamente el padrón al alumno que le corresponde (el siguiente del último en existencia).

```python
def add(self, session: Session, alumno_a_crear: AlumnoUpsert) -> Alumno:
    alumno = Alumno(
        nombre=alumno_a_crear.nombre,
        apellido=alumno_a_crear.apellido,
        edad=alumno_a_crear.apellido
    )
    session.add(alumno)
    session.commit()
    session.refresh(alumno)
    return alumno
```

Todo esto se puede escribir en una línea:

```python
alumno = Alumno(**alumno_a_crear.model_dump())
```

Siempre que queramos hacer una modificación a lo que hay guardado en la DB tenemos que hacer estos 3 pasos:

Agregarlo a la sesión

Hacer el commit en la DB para que se reflejen los cambios

Actualizar el modelo con los campos autogenerados (si no, no tendríamos padrón).

Add

Ejercicio (para programar un poco si hay tiempo)

Al crear un alumno, se le debe asignar automáticamente su email. El mismo se generará a partir de la primera letra de su nombre + su apellido + "@fi.uba.ar".

Extras

- Si el alumno tiene un nombre compuesto, agregar la primer letra de cada uno de sus nombres.
- Si tiene un apellido compuesto, concatenarlos.
- Si el email ya existiese, agregar la siguiente letra de su/s nombre/s. Repetir hasta que no exista el email.

Update

```python
def update(self, session: Session, padron: int, nuevo_alumno: AlumnoUpsert) -> Alumno:
    alumno = self.find(session, padron)
    alumno.nombre = nuevo_alumno.nombre
    alumno.apellido = nuevo_alumno.apellido
    alumno.edad = nuevo_alumno.edad

    session.add(alumno)
    session.commit()
    session.refresh(alumno)
    return alumno
```

1. Buscamos al alumno existente por padrón
2. Copiamos todos los campos del alumno que recibimos al que encontramos.
3. Los mismos 3 pasos del `create` que tenemos que hacer cada vez que modificamos una row de la DB.

OJO si el cliente manda algún campo en None. Para los campos que no querramos actualizar en ese caso, deberíamos chequear primero que no sea None.

Update

```python
def update(self, session: Session, padron: int, nuevo_alumno: AlumnoUpsert) -> Alumno:
    alumno = self.find(padron)
    update_dict = nuevo_alumno.model_dump(exclude_unset=True)
    alumno.sqlmodel_update(update_dict)

    session.add(alumno)
    session.commit()
    session.refresh(alumno)
    return alumno
```

Ignora los campos nulos.
`nuevo_alumno` es de tipo `AlumnoUpsert` (igual a `AlumnoBase`) sin padrón para no actualizar el padrón sin querer.

Delete

```python
def delete(self, session: Session, padron: int) -> Alumno:
    alumno = self.find(padron)
    session.delete(alumno)
    session.commit()
    return alumno
```

Manejo de la sesión

Por limitaciones del framework no podemos usar dependencias en métodos que no definan endpoints.

Manejar la sesión desde la clase Database/Repository puede quedar medio incómodo o traer errores muy fácilmente.

Recomendamos seguir los lineamientos establecidos en estas diapos que seguro funcionan. Los mismos implican tener que agregar un poco de código a los endpoints que ya hicimos pero no es muy grave.

Tests

¿Cómo hacemos para probar la conexión y las queries con la DB?

Podemos crear una DB "de mentiritas" en memoria que sólo va a existir para los tests. Tenemos un par de facilidades para lograr esto provistas por el ORM y el framework de testing.

Test - 1er paso: engine fixture

Debemos agregar este fixture en algún módulo común para nuestros tests. Proveerá del engine para crear las sesiones de nuestros tests. La DB creada vive en memoria. Es destruída automáticamente luego de cada test.

```
@pytest.fixture()
def engine():
    engine = create_engine(
        "sqlite://", connect_args={"check_same_thread": False}, poolclass=StaticPool
    )
    SQLModel.metadata.create_all(engine)
    return engine
```

En vez de migraciones, los tests crean las tablas en base a los modelos definidos.

Recordando.... Un *fixture* en este contexto no es más que una función inicializadora de algo.
pytest nos permite definir fixtures con annotations, los cuales se van a correr previo a los test que vayamos a correr y van a estar disponibles para todos ellos.

Tests - 2do paso: session fixture

Más de lo mismo, pero para las session. Ojo que tienen una parte especial, yield.

```python
@pytest.fixture
def session(engine) -> Generator[Session, None, None]:
    with Session(engine) as session:
        yield session
```

El engine que va a usar es el que definimos en el fixture anterior, lo detecta automáticamente el framework

Tests - 3er paso: db fixture

Casi el mismo que ya teníamos de antes

```
@pytest.fixture
def db():
    return Database()
```

Tests - conftest

Podemos agrupar los fixtures que necesitemos en un sólo lugar

Si tuviéramos varias clases que necesitan de los mismos fixtures (por ejemplo si tuviéramos múltiples clases de DB) podemos incluirlos en un archivo

`conftest.py`

y los mismos se van a incluir automáticamente en todos los tests en el directorio donde se encuentre el mismo.

Tests - 4to paso: hacer lo tests

Cada test que hagamos recibirá por parámetro el fixture de la DB

```
def test_list_no_vacio(self, session, db):
    al1 = Alumno(nombre="Nombre1", apellido="Apellido1", edad=100)
    al2 = Alumno(nombre="Nombre2", apellido="Apellido2", edad=99)
    session.add(al1)
    session.add(al2)
    session.commit()
    session.refresh(al1)
    session.refresh(al2)

    resultado = db.list(session)

    assert len(resultado) == 2
    assert resultado == [al1, al2]
```

setup: cargamos todos los datos que necesitamos.

execute: hacemos la llamada al método/función que queremos testear

verify: verificamos que el resultado de la llamada sea el esperado

Tests - 4to paso: hacer lo tests (alternativa)

En vez de usar la session directamente, podemos usar los métodos de la DB

```
def test_list_no_vacio(self, session, db):
    al1 = Alumno(nombre="Nombre1", apellido="Apellido1", edad=100)
    al2 = Alumno(nombre="Nombre2", apellido="Apellido2", edad=99)
    al1 = db.add(session, al1)
    al2 = db.add(session, al2)

    resultado = db.list(session)

    assert len(resultado) == 2
    assert resultado == [al1, al2]
```

setup: cargamos todos los datos que necesitamos.

execute: hacemos la llamada al método/función que queremos testear

verify: verificamos que el resultado de la llamada sea el esperado

Tests: actualizar los tests de los routers

Ahora que tenemos una nueva dependencia hay que actualizar los tests de los routers también

Depende cómo estén hechos, habrá que actualizar más o menos cosas.
Los tests deberían estar usando un mock para la DB, pero además, para ser aún más completos, deberían estar chequeando que en ese mock se llamen a los métodos correctos con los valores correctos.
Por ejemplo, en un test de `get_alumnos` deberíamos tener

```
mock_db.list.assert_called_once()
```

pero ahora que le pasamos argumentos al mock cuando lo llamamos debemos cambiar la verificación por algo así

```
mock_db.list.assert_called_once_with(mock_session, FiltrosAlumno())
```

Y agregar el `mock_session` igual que el `mock_db`

```
mock_session = MagicMock(Session)
app.dependency_overrides[get_session] = lambda: mock_session
```

No deberían haber más cambios que esos

Tests - qué hay que probar?

Los casos a testear van a depender del tipo de endpoint que estemos testeando (justamente), pero hay ciertas cosas generales que aplican para todos:

- todo flujo que pueda dar un error (por cada error distinto deberíamos tener un test)
- todo flujo que sea exitoso usando distintos parámetros (si al crear una nueva entidad pasan cosas distintas si le paso el parámetro A o el B; si al listar hay suficientes elementos, o ninguno, o uso filtros o paginado; etc.

Además, en cada caso de prueba necesitamos validar correctamente el resultado de la operación. Esto implica:

- ver que la response del servidor sea correcta (tenga el status correcto)
- ver que el contenido del body de la response sea el correcto (que tenga los elementos esperados, en el orden esperado, etc.)

Tablas Relacionadas

Tablas Relacionadas

Podemos referenciar fácilmente tablas relacionadas por foreign keys.

Además de tener las tablas con las correspondientes constraints, para facilitar el manejo de estas relaciones, SQLModel nos introduce un nuevo tipo de campo para nuestros modelos: `Relationship`.

Relación 1:N: configuración inicial

Creemos una nueva migración que nos agregue una nueva tabla `grupo` y además vincule un alumno con un grupo a través de una foreign key.

ANTES DE HACER ESO debemos agregar una línea al archivo `env.py` de Alembic:

```
context.configure(
    ...
    render_as_batch=True,
)
```

Sólo necesitamos agregar este argumento, el por qué

Ahora sí, tenemos que agregar la nueva tabla y la foreign key.

Para ser prolijos, vamos a hacerlo en dos migraciones distintas.

Relación 1:N: migración

Nos conviene hacer dos migraciones distintas, simplemente para mantener las cosas más ordenadas y meter de a un cambio por vez

Primero creamos la nueva tabla

```python
def upgrade() -> None:
    op.create_table(
        'grupo',
        sa.Column('id', sa.Integer, primary_key=True),
        sa.Column('nombre', sa.Text, nullable=False),
    )

def downgrade() -> None:
    op.drop_table('grupo')
```

Luego creamos la constraint

```python
def upgrade() -> None:
    with op.batch_alter_table('alumno') as batch_op:
        batch_op.add_column(sa.Column('grupo_id', sa.Integer))
        batch_op.create_foreign_key('fk_alumno_grupo', 'grupo', ['grupo_id'], ['id'])

def downgrade() -> None:
    with op.batch_alter_table('alumno') as batch_op:
        batch_op.drop_constraint(
            'fk_alumno_grupo', type_='foreignkey')
        batch_op.drop_column(
            'grupo_id'
        )
```

Si queremos hacer migraciones que modifiquen una tabla hay que agregar este "`batch_op`"

Migraciones

Toda migración tiene su propio identificador y el de su migración padre.

Primer migración, no tiene padre

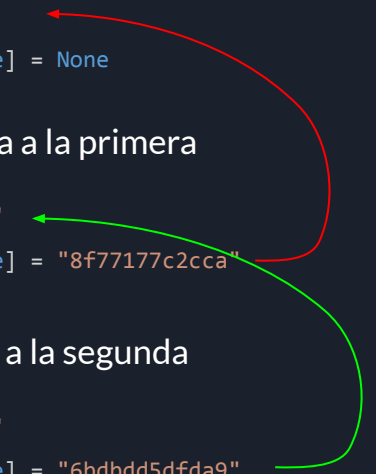
```
revision: str = '8f77177c2cca'
down_revision: Union[str, None] = None
```

Segunda migración, apunta a la primera

```
revision: str = "6bdbdd5dfda9"
down_revision: Union[str, None] = "8f77177c2cca"
```

Tercera migración, apunta a la segunda

```
revision: str = "9e6f41a4bd49"
down_revision: Union[str, None] = "6bdbdd5dfda9"
```



Es importante que se mantenga el orden lineal.

Cada migración tiene exactamente un padre y es padre de exactamente otra migración.

A veces hay herramientas que ayudan a chequear que esto no se rompa (por ejemplo, hay 2 pull request al mismo tiempo que ambos agregan una migración con el mismo padre).

En este caso, no, hay que hacerlo a mano.

Relación 1:N: modelos

```python
class Alumno(AlumnoBase, table=True):
    padron: int = Field(primary_key=True)

    grupo_id: int | None = Field(default=None, foreign_key="grupo.id")
    grupo: Grupo | None = Relationship(back_populates="alumnos")

class Grupo(SQLModel, table=True):
    id: int = Field(primary_key=True)
    nombre: str

    alumnos: list["Alumno"] = Relationship(back_populates="grupo")
```

Alumno está entre " porque se referencia la clase antes de tenerla definida. Entonces usa el nombre de la clase en un `string`

Relación 1:N: modelos

```
class Alumno(AlumnoBase, table=True):
    padron: int = Field(primary_key=True)

    grupo_id: int | None = Field(default=None, foreign_key="grupo.id")
    grupo: Grupo | None = Relationship(back_populates="alumnos")
```

`grupo_id` es la foreign key. Tiene únicamente el ID numérico del modelo relacionado. Debemos especificar el nombre de la tabla relacionada y su campo al cual hace referencia. (tabla grupo y el campo id).

Por defecto es None porque un alumno puede no tener un grupo.

El campo grupo es una construcción del framework que estamos usando, una facilidad que nos provee. Normalmente, dado un alumno, para ver a qué grupo pertenece deberíamos hacer una query de este estilo:

```
session.exec(select(Grupo).where(Grupo.id == alumno.grupo_id)).first()
```

Gracias a este campo, podemos simplemente hacer

```
alumno.grupo
```

Relación 1:N: modelos

```python
class Grupo(SQLModel, table=True):
    id: int = Field(primary_key=True)
    nombre: str

    alumnos: list["Alumno"] = Relationship(back_populates="grupo")
```

`back_populates`: sirve para que SQLModel entienda a qué campo del otro modelo hace referencia. Para así, si se modifica el objeto, sabe qué campo modificar del objeto relacionado.

Alumno está entre " porque se referencia la clase antes de tenerla definida. Entonces usa el nombre de la clase en un `string`

Relación 1:N: database

Una alternativa es tener una clase DB por cada tipo de modelo que tengamos definido. De esta manera separamos las responsabilidades de cada una. Pero va a haber situaciones en las que en un mismo flujo tengamos que consultar por varios modelos distintos, si fuera una sola clase sería más fácil y podemos reutilizar más cosas.

Pueden plantear cualquier alternativa, u otras si quisieran (siempre y cuando tengan sentido). En los siguientes ejemplos voy a usar una DB para cada modelo.

Por lo que entonces necesitamos definir una nueva DB para grupos, instanciarla, tener la dependencia y todo lo demás que ya hicimos. Además, debemos cambiarle el nombre a la que ya teníamos para que sea más claro que es para alumnos.

Entonces necesitamos un método en la db de Grupos para hacer un find de un grupo, y un método en la DB de alumnos para agregarselo (el find es necesario para comprobar que el id de grupo que se envía en la request es un ID válido).

Relación 1:N: database

```python
class DBGrupos:
    def find(self, session: Session, grupo_id: int) -> Grupo:
        grupo = session.exec(select(Grupo).where(Grupo.id == grupo_id)).first()
        if grupo:
            return grupo
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND, detail="Grupo not found"
        )

class DBAlumnos:
    def inscribirse_a_grupo(self, session: Session, padron: int, grupo: Grupo) -> Alumno:
        alumno = self.find(padron)
        alumno.grupo = grupo
        session.add(alumno)
        session.commit()
        session.refresh(alumno)
        return alumno
```

Relación 1:N: database

```python
class DBGrupos:
    def find(self, session: Session, grupo_id: int) -> Grupo:
        grupo = session.exec(select(Grupo).where(Grupo.id == grupo_id)).first()
        if grupo:
            return grupo
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND, detail="Grupo not found"
        )

class DBAlumnos:
    def inscribirse_a_grupo(self, session: Session, padron: int, grupo_id: int) -> Alumno:
        alumno = self.find(padron)
        alumno.grupo_id = grupo_id
        session.add(alumno)
        session.commit()
        session.refresh(alumno)
        return alumno
```

Relación 1:N: endpoint

Donde hacemos el endpoint?

`/alumnos/{padron}/asignar_grupo`

`/grupos/{grupo_id}/agregar_integrante`

Si lo que vamos a editar es un Alumno, tiene más sentido que sea parte de las rutas de Alumno, pero ambas pueden ser.

```python
@router.put("/{padron}/asignar_grupo", status_code=status.HTTP_200_OK)
def asignar_grupo(
    session: SessionDep,
    db_alumnos: DBAlumnosDep,
    padron: int,
    grupo_id: int,
) -> Alumno:
    return db_alumnos.inscribirse_a_grupo(session, padron, grupo_id)
```

Necesitamos también definir las rutas de Grupo (listar, mostrar, crear) que hasta ahora no hicimos.

Mostrar objetos relacionados

Por defecto, al mostrar un alumno, sólo veremos el grupo_id y no la información del grupo.

De la misma manera al mostrar un grupo, no veremos los alumnos asignados al mismo.

Para ellos debemos cambiar ligeramente el modelo que se devuelve.

```python
class GrupoPublic(GrupoBase):
    id: int
    alumnos: list["Alumno"] = []
```

```python
class AlumnoPublic(AlumnoBase):
    padron: int
    grupo: Grupo | None = None
```

Creamos nuevas clases que NO sean tablas

```python
def show(grupo_id: int) -> GrupoPublic:
```

```python
def show(padron: int) -> AlumnoPublic:
```

Modificamos los endpoints para que devuelvan estos nuevos tipos.

Mostrar objetos: un paso más

Es buena práctica separar los modelos que maneja la DB de los modelos que se manejan en los endpoints. Hay representaciones internas que se pueden querer ocultar o cambiar como se muestran. Además, si usamos modelos específicos para los endpoints, podemos agregarle los campos que necesitemos.

```python
class AlumnoPublic(AlumnoBase):
    padron: int

class GrupoPublic(GrupoBase):
    id: int

class AlumnoPublicWithRelations(AlumnoPublic):
    grupo: GrupoPublic | None = None

class GrupoPublicWithRelations(GrupoPublic):
    integrantes: list[AlumnoPublic] = []
```

```python
def list() -> GrupoPublic:
def show(grupo_id: int) -> GrupoPublicWithRelations:

def list() -> AlumnoPublic:
def show(padron: int) -> AlumnoPublicWithRelations:
```

Hasta podemos modificar `GrupoPublicWithRelations` para que muestre la lista de alumnos solo con el padrón, o sólo el nombre, etc.

Relación N:M

Vamos a complejizar un poco lo que armamos, para que sea más realista. A partir de ahora, un alumno puede tener múltiples grupos de tp, ya que para cada entrega podría tener uno distinto, por ejemplo. Pero además, queremos que cada integrante de cada tp pueda tener una nota distinta en cada TP (según su participación, etc.)

Cada alumno puede tener múltiples grupos, cada grupo múltiples alumnos. Necesitamos una tabla intermedia para representar esta relación.

Y las notas dónde las guardamos?

En la tabla intermedia vamos a tener una columna adicional para guardar esta información

Relación N:M: migraciones

Creemos la tabla Integrante que tendrá referencias a alumno y al grupo. Y además, tendrá la nota del alumno para ese grupo.

Luego, debemos modificar la tabla de alumnos para que ahora no tenga una fk a grupo, sino que pase por la tabla intermedia que acabamos de crear.

```python
op.create_table(
    "integrante",
    sa.Column('nota', sa.Integer),
    sa.Column('alumno_padron', sa.Integer, primary_key=True),
    sa.Column('grupo_id', sa.Integer, primary_key=True),
    sa.ForeignKeyConstraint(
        ["alumno_padron"], ["alumno.padron"],
    sa.ForeignKeyConstraint(
        ["grupo_id"], ["grupo.id"],
    ),
)
```

`alumno_padron` y `grupo_id` son una `primary_key` compuesta, y además cada una es una `foreign_key`

No se olviden los downgrade también!

Es la opuesta a la que hicimos la vez pasada

```python
def upgrade() -> None:
    with op.batch_alter_table("alumno") as batch_op:
        batch_op.drop_constraint("fk_alumno_grupo", type_="foreignkey")
        batch_op.drop_column("grupo_id")
```

Relación N:M: modelos

```python
class IntegranteBase(SQLModel):
    nota: int | None = Field(default=None, ge=1, le=10)


class Integrante(IntegranteBase, table=True):
    grupo_id: int = Field(
        nullable=False, foreign_key="grupo.id", primary_key=True
    )
    grupo: "Grupo" = Relationship(back_populates="integrantes")

    alumno_padron: int = Field(
        nullable=False, foreign_key="alumno.padron", primary_key=True
    )
    alumno: "Alumno" = Relationship()
```

Eso es suficiente para representar la relación, pero al ORM nos da una facilidad más para poder conseguir los modelos relacionados desde un `Alumno` o `Grupo`.

Relación N:M: modelos

```python
class Grupo(SQLModel, table=True):
    id: int = Field(primary_key=True)

    integrantes: list[Integrante] | None = Relationship(back_populates="grupo")

class Alumno(AlumnoBase, table=True):
    padron: int = Field(primary_key=True)

    grupos: list[Grupo] | None = Relationship(back_populates="alumnos", link_model=Integrante)
```

Grupo solo le interesa relacionarse con integrantes, no con los alumnos.

Alumno le interesa relacionarse con los grupos, a traves de Integrante.

Pero no usamos back_populates en este caso. Como la relación no es recíproca, no debemos usar ese modificador.

Relación N:M: endpoints

Veamos primero qué queremos hacer, qué acciones queremos poder tomar con nuestros alumnos y grupos

- Sigo queriendo poder crear un grupo vacío pasandole sólo un nombre
- Sigo queriendo poder inscribir un alumno a un grupo.
- Cuando muestro un grupo, quiero mostrar sus alumnos/integrantes (con la nota de cada uno)
- Quiero ahora además mostrar todos los grupos de un alumno y la nota que tiene en cada uno.
- Quiero ahora además cargarle una nota a un alumno que ya está en un grupo particular.
- Quiero poder crear un grupo ya pasandole los integrantes y sus notas.

Inscribir un alumno a un grupo

Podemos modificar lo que ya teníamos para incluir la creación de un integrante

Cuando la relación era 1:N simplemente asignabamos el grupo_id al alumno y listo.

Ahora hay un modelo intermedio (Integrante) que representa esa relación. Por lo tanto debemos crear uno apuntando al alumno y al grupo correspondiente.

```
def inscribirse_a_grupo(self, session: Session, padron: int, grupo_id: int) -> Alumno:
    integrante = Integrante(alumno_padron=padron, grupo_id=grupo_id)
    session.add(integrante)
    session.commit()
    alumno = self.__get(session, padron)
    return alumno
```

Podemos recibir el grupo o recibir su ID.
Es preferible recibir el ID directamente para no tener que hacer otra query previamente.

Mostrar las nuevas relaciones

Definimos los modelos públicos igual a como veníamos haciendo y los devolvemos en los endpoints

```python
class IntegrantePublic(IntegranteBase):
    pass

class AlumnoPublicWithRelations(AlumnoPublic):
    grupos: list[GrupoPublic] = []

class IntegrantePublicWithAlumno(IntegrantePublic):
    alumno: AlumnoPublic

class IntegrantePublicWithRelations(IntegrantePublic):
    alumno: AlumnoPublic
    grupo: GrupoPublic

class GrupoPublicWithIntegrantes(GrupoPublic):
    integrantes: list[IntegrantePublicWithAlumno] = []
```

```
Alumnos
def show(...) -> AlumnoPublicWithRelations
def asignar_grupo(...) -> AlumnoPublicWithRelations

Grupos
def show(...) -> GrupoPublicWithIntegrantes
def list(...) -> list[GrupoPublic]
```

Cargarle una nota en un grupo a un alumno

Deberíamos definir un nuevo combo de `router+db` para el nuevo modelo integrante.

```python
@router.put("/{grupo_id}/{padron}/cargar_nota")
def cargar_nota(
    session: SessionDep, db: DBIntegrantesDep,
    grupo_id: int, padron: int, nota: int
) -> IntegrantePublicWithRelations:
    integrante = db.poner_nota(session, grupo_id, padron, nota)
    return integrante
```

Integrante no tiene un ID propio (podría tenerlo pero no lo definimos) por lo que la forma de identificarlos es a través de los IDs de grupo y alumno.

```python
def poner_nota(self, session: Session, grupo_id: int, padron: int, nota: int) ->
Integrante:
    integrante = session.exec(
        select(Integrante)
        .where(Integrante.grupo_id == grupo_id)
        .where(Integrante.alumno_padron == padron)
    ).first()

    if not integrante:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND, detail="Integrante not found"
        )

    integrante.nota = nota
    session.add(integrante)
    session.commit()
    session.refresh(integrante)
    return integrante
```

Crear un grupo con integrantes y notas

El endpoint de crear grupo debería poder recibir toda la información necesaria para poder crear un grupo ya con integrantes y cada uno de ellos con su nota.

```
@router.post(
    "/",
    status_code=status.HTTP_201_CREATED,
    responses={status.HTTP_422_UNPROCESSABLE_ENTITY: {"model": Error}},
)
def create(session: SessionDep, db: DBGruposDep, grupo_a_crear: GrupoUpsert) ->
GrupoPublicWithIntegrantes:
    if len(grupo_a_crear.integrantes) > MAX_SIZE:
        raise HTTPException(
            status_code=status.HTTP_422_UNPROCESSABLE_ENTITY,
            detail="Grupo admite hasta 4 integrantes",
        )
    grupo = db.add(session, grupo_a_crear)
    return grupo
```

Devolvemos los integrantes también, ya ahora puede tener al momento de crearse.

No se olviden de actualizar GrupoUpsert para que acepte Integrantes también

```
class GrupoUpsert(GrupoBase):
    integrantes: list[Integrante] | None
```

Crear un grupo con integrantes y notas

El endpoint de crear grupo debería poder recibir toda la información necesaria para poder crear un grupo ya con integrantes y cada uno de ellos con su nota.

```python
def add(self, session: Session, grupo_a_crear: GrupoUpsert) -> Grupo:
    grupo = Grupo(nombre=grupo_a_crear.nombre)
    session.add(grupo)
    integrantes = [
        Integrante(
            grupo=grupo,
            alumno_padron=integrante.alumno_padron,
            nota=integrante.nota,
        )
        for integrante in grupo_a_crear.integrantes
    ]
    session.add_all(integrantes)
    session.commit()
    session.refresh(grupo)
    return grupo
```

Creamos el grupo primero, y luego los integrantes.

El ORM debería ser lo suficientemente inteligente para que si hacemos

```python
grupo = Grupo(**grupo_a_crear.model_dump())
```

debería poder crear el grupo y todos los integrantes sin tener que hacerlos por separado.

Pero a mi no me anduvo, si les anda avisenme.

Seeds

Cómo cambian nuestras seeds ahora que tenemos una DB que persiste?

Antes corríamos las seeds siempre al comienzo del programa, porque no se mantenía el estado. Si hacemos lo mismo ahora, estaríamos duplicando un montón de data en cada corrida.

Debemos tener lógica a mano para correr las seeds sólo en determinadas circunstancias.

Hay muchas opciones, algunas son:

- Seguir teniendo un script similar al que ya teníamos pero correrlo únicamente si detectamos que la DB está vacía.
- Tener una migración que llene las tabla con los datos.
- Tener un script aparte que pueda correr desde la terminal cuando yo quiera

Seeds: función al iniciar servidor

La lógica es muy parecida a la actual, en nuestro main vamos a llamar a la función del módulo seed.py para iniciar la lógica.

La diferencia está en que lo vamos a hacer sólo si detectamos que no hay nada cargado en la DB previamente:

```
with Session(engine) as session:
    # Do no seed if there's data present in the DB
    if session.exec(select(Alumno)).first():
        return
    ...
```

Seeds: migración

Es un poco más prolijo, pero estamos limitados a que se van a correr únicamente cuando se hagan las migraciones.

```
root = os.path.abspath(os.path.join(os.path.dirname(__file__), "../.."))

def upgrade() -> None:
    path = os.path.join(root, "resources", "alumnos.csv")
    alumnos = []
    with open(path) as f:
        ...

    alumnos_table = sq.table(
        "alumno",
        sq.Column("padron", sq.Integer, primary_key=True),
        sq.Column("nombre", sq.Text, nullable=False),
        sq.Column("apellido", sq.Text, nullable=False),
        sq.Column("edad", sq.Integer),
    )

    op.bulk_insert(alumnos_table, alumnos)
```

Hay que definir la tabla de nuevo, por más que ya la teníamos antes

No hay que olvidarse el downgrade también. En este caso tendría que eliminar todos los registros de la tabla alumno.

Seeds: script separado

Es la opción más maleable de las tres, pero es propensa a errores.

Es común definir un directorio donde tener scripts diversos y poner nuestra lógica allí

```
scripts/
  seeds.py
```

Podemos separarlo también en múltiples scripts si queremos tener la posibilidades de cargar data por partes.
Luego es simplemente cuestión de correr en la terminal

```
python scripts/seeds.py
```

cuando querramos.
Chequeen bien antes de hacerlo, no hay ningún chequeo automático presente para detectar si ya hay datos. Pueden estar generando duplicados sin darse cuenta.

Seeds con modelos relacionados

Ahora que tenemos modelos que se relacionan entre sí, si queremos hacer seeds en nuestra DB, debemos hacerlo en cierto orden para que se respeten las foreign key establecidas (ej: no puedo crear un Integrante si no existe el Alumno y el Grupo a los cuales lo quiero asignar).

No hay una fórmula que nos permita siempre resolver esto. Hay que tener presente las relaciones e ir creando los modelos según estas dependencias.

En nuestro caso deberíamos crear en este orden: **`Alumnos`** -> **`Grupos`** -> **`Integrantes`**.

Si hay más relaciones complejas el orden puede no ser tan claro.

Bonus Tracks

Migraciones

Cómo sabe el sistema cuál fue la última migración aplicada?

La mayoría de las bibliotecas que manejan migraciones lo hacen guardando la información en la base de datos.

Si ven la info de la DB luego de setupear y correr Alembic verán una tabla llamada `alembic_versions` que tiene una sola columna y una sola fila con el ID de la migración más reciente. Al hacer un downgrade or upgrade se actualizará este dato.

Fin